# Xamarin with Visual Studio

Launch your mobile development career by creating Android and iOS applications using.NET and C#

ALESSANDRO DEL SOLE

# Xamarin with Visual Studio

Launch your mobile development career by creating Android and iOS applications using.NET and C#

ALESSANDRO DEL SOLE

bpb

# Xamarin with Visual Studio

*Launch Your Mobile Development Career by Creating Android and iOS Applications Using.NET and C#*

**Alessandro Del Sole**

www.bpbonline.com

**Group Product Manager:** Marianne Conor

**Publishing Product Manager:** Eva Brawn

**Senior Editor:** Connell

**Content Development Editor:** Melissa Monroe

**Technical Editor:** Anne Stokes

**Copy Editor:** Joe Austin

**Language Support Editor:** Justin Baldwin

**Project Coordinator:** Tyler Horan

**Proofreader:** Khloe Styles

**Indexer:** V. Krishnamurthy

**Production Designer:** Malcolm D'Souza

**Marketing Coordinator:** Kristen Kramer

**First published:** July 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-55511-874

**www.bpbonline.com**

## Dedicated to

*My wonderful wife* ***Angelica****, I'll love you until the end*

# About the Author

**Alessandro Del Sole** is a senior software engineer working on building mobile apps with Xamarin in the healthcare industry. He has been using Xamarin since 2015 and is a Xamarin Certified Mobile Developer. Alessandro has also been Microsoft Most Valuable Professional since 2008, and has authored many technical books, eBooks, articles, instructional videos and has been speaker in the most important Italian conferences. He has also been recently appointed as C# Corner MVP.

# About the Reviewer

**Ninaada** is a Xamarin certified mobile developer with 10+ years of industry experience. He has extensively worked on Xamarin based projects for more than 6 years, across various domains and industries including Fortune 500 companies. He also is active in the xamarin community by contributing to GitHub and blogging occasionally. Currently, he is working as a Xamarin developer with Steer73. He also has a keen interest in nature and wildlife photography, which can be found on his social media pages. Find more at **https://linktr.ee/ninaada**

# Acknowledgement

I would first like to thank so much my wife, Angelica, who respects the time I need to write and that never stops to be supportive to me, always demonstrating to me so much love.

I would like to then thank the team at BPB Publications for being so supportive. I have been authoring books for 15 years, but this was my first time with them, and the whole team made it simpler for me to approach to a different way of working.

Special thanks to the amazing Arielle (**https://imarielle.com**), whose music has been my fuel during the many hours I spent writing.

Finally, I would like to thank all the readers of this book and all those who have read other books or articles I wrote in the past years. Writing is a big effort, so you need a motivation to do it and having had good feedback over the years, is the right motivation for me to still write.

# Preface

This book covers mobile app development from the point of view of the developer that wants to target both iOS and Android systems from one codebase, using development tools from Microsoft.

The book is intended for both the experienced and the novice developers, with a special, practical approach for those who want to start a new career as software developers in the mobile apps industry.

Basic knowledge of the C# programming language is recommended, however the book introduces general programming and language-specific concepts that will be used across the book, so that also the novice developer can be familiar with all the discussed topics.

The goal of the book is explaining how to build mobile apps from the ground up, focusing on all the necessary aspects: the user interface, the logic, the interaction with Web services, the consumption of data, the way an app should be architected for mobile devices.

Special focus is also offered for those who want to start searching a new occupation as software developers with a Microsoft technical background.

This book is divided into **15 chapters**. The first four chapters introduce the reader into the world of software development with C#, Microsoft Visual Studio and .NET. Chapters from 5 to 13 are all technical chapters that cover the development of mobile apps with Xamarin, in a cross-platform approach. The last two chapters have specific focus on starting, building, and maintaining a career as a successful mobile developer. Below you find the details of each chapter.

**Chapter 1** describes why it is important to be a mobile app developer today, and why building apps in a cross-platform approach is the best choice especially for jobseekers.

**Chapter 2** describes how Microsoft jumped into the mobile app development ecosystem. An introduction to Xamarin is made, plus a you find a discussion about why choosing Xamarin as the development platform can be the key to success today.

**Chapter 3** gives an introduction on the .NET technology (including the latest updates) and to Microsoft Visual Studio as a development environment

**Chapter 4** is intended to give absolute beginners a quick overview of the C# programming language to make them feel more comfortable with the development tools used across the book.

**Chapter 5** describes how to create a Xamarin solution with Microsoft Visual Studio, explaining how the code sharing strategy works to deliver apps to multiple OSes from the same codebase.

**Chapter 6** explains the fundamentals of user interface in mobile apps, and how to create adaptive and auto-sizing user interfaces using layouts.

**Chapter 7** chapter describes how to create the user interface of an app with the controls offered by Xamarin and how to manage user interaction with controls.

**Chapter 8** describes how to organize the user interface of an app in pages, the different kinds of pages and how navigation between pages works.

**Chapter 9** describes the concept of reusable resource and explains how to connect data objects to the user interface for automatic communication. It also explains how to use local SQLite databases to store, retrieve and display data.

**Chapter 10** discusses how to enrich the user interface of an app with brushes, shapes, and with media contents.

**Chapter 11** explains how to handle the different events of an app's lifecycle and how to manage common requirements of mobile apps, such as network connection, battery level, local settings.

**Chapter 12** describes how to consume Web services in Xamarin projects, including Cloud services hosted on Azure.

**Chapter 13** describes how to access native iOS and Android APIs from Xamarin, pointing out how it is something to do only when the Xamarin and Xamarin.Essentials codebases do not have something that works cross-platform. It also describes how to implement customizations that are platform-specific.

**Chapter 14** helps you understand how to search for jobs that are related to mobile app development with Xamarin by making focused, appropriate job searches based on the reader's experience after understanding the core business of a company that is hiring.

**Chapter 15** provides advice on what a good approach is in order to keep a job in the mobile app development market including tips & tricks to work in a team and to stay up to date with technology.

# Code Bundle and Coloured Images

Please follow the link to download the
***Code Bundle*** and the ***Coloured Images*** of the book:

## **https://rebrand.ly/ziayzy4**

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Xamarin-with-Visual-Studio**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

# Table of Contents

**1. The Importance of Mobile App Development**
- Introduction
- Structure
- Objectives
- The importance of mobile app development
  - *Mobile devices and applications*
    - *A brief history of modern mobile devices*
    - *Mobile devices for everyone*
    - *Mobile devices everywhere*
  - *The importance of mobile apps for companies*
  - *Career opportunities with mobile app development*
    - *Making money with mobile app development*
  - *Summarizing how a mobile system works*
- The importance of a cross-platform approach
  - *Limitations of cross-platform code*
  - *Application size and performance*
    - *Understanding native applications*
    - *Demystifying application performance issues*
- Staying up to date with new devices and platforms
  - *Getting updates from Apple*
  - *Getting updates from Google*
  - *Getting news from Microsoft*
- Conclusion
- Points to remember

**2. Xamarin and Microsoft in the Mobile App Market**
- Introduction
- Structure
- Objectives
- Xamarin as the app development framework
  - *Mobile App Development with Microsoft Skills*
  - *Xamarin as a company*

*Introducing the mono project*
*The transition to Xamarin*
*Microsoft in the mobile app ecosystem*
*The importance of Microsoft investments on Xamarin*
*Improved productivity with Microsoft tools and services*
Understanding Xamarin and Xamarin.Forms
*Advantages of Xamarin.Forms*
*Hints about .NET MAUI*
Installing and configuring the development tools
*System requirements for Xamarin*
*System requirements on Windows*
*System requirements on macOS*
*Android and iOS Devices*
*Installing and configuring Microsoft Visual Studio*
*Selecting workloads*
*Checking for individual components*
*Configuring Android devices for developer mode*
*Minimum macOS configuration*
Conclusion
Questions
Points to remember

**3. Introducing .NET and Visual Studio**
Introduction
Structure
Objectives
Introducing .NET
*Introducing the .NET framework for Windows*
*Locating the .NET framework on disk*
*The .NET framework architecture*
*The Common Language Runtime (CLR)*
*Compilers and the concept of Assembly*
*Assemblies in .NET*
*The Base Class Library*
*From .NET Framework to .NET Core*
*Advantages of .NET Core*
.NET 6: One .NET

Building applications with Visual Studio
*Signing in with a Microsoft account*
*Synchronized settings*
*Introducing the Start window*
*Understanding projects and solutions*
*Creating projects with C#*
*Creating your first C# project*
Working with the Visual Studio IDE
*Working with tool windows*
*Solution Explorer*
*The Error List*
*The Properties window*
*The Output window*
*Editing project properties*
*Editing application settings*
*Basic code editing features*
*Syntax colorization*
*Zooming the code editor*
*Introducing IntelliSense*
*Changing the Code Editor options*
Compiling, running, and debugging code
*Understanding configurations*
*Running and Debugging Code*
*Debugging your code*
*Preparing the sample project for debugging*
*Breakpoints and data tips*
*Detecting and fixing runtime errors*
Conclusion
Points to remember
Key terms

**4. The C# Programming Language**
Introduction
Structure
Objectives
Understanding data types
*Meet the Common Type System*

    - *Clarifying value types and reference types*
    - *.NET primitive types*
        - *Declaring and consuming variables*
        - *Clarifying the difference between value and reference types*
- Common data operators
    - *Equals and Not Equals Operators*
    - *Arithmetic operators*
    - *Conversion operators*
        - *Direct type conversion*
        - *Converting types with the Convert class*
        - *String conversion*
    - *Logical operators*
    - *Working with strings*
        - *Checking for Null or Empty strings*
        - *Concatenating strings*
        - *Formatting strings*
- Iterating objects
- Understanding loops
    - *Introducing the do loop*
    - *Introducing the while loop*
- Conditional code blocks
    - *The if conditional code block*
    - *The switch conditional code block*
- Introducing arrays
    - *Single-dimensional arrays*
    - *Multi-dimensional arrays*
    - *Jagged arrays*
- Object-oriented programming (OOP)
    - *Understanding access modifiers*
    - *Defining reference types with classes*
        - *Storing information with fields and properties*
        - *Running actions with methods*
        - *Introducing constructors and static classes*
        - *Defining derived types with inheritance*
    - *Understanding interfaces and abstract classes*
        - *Defining interfaces*
        - *Implementing interfaces*

*Introducing abstract classes*
*Organizing types within namespaces*
*Accessing types within namespaces with using directives*
*Defining value types with structures*
*Assigning structures to variables*
*Visibility of structures and their members*
*Defining enumerations*
*Using enumerations*
*Implementing error handling*
*Implementing events*
Advanced C# programming
*Generics and Nullable Types*
*Working with Nullable types*
*Language INtegrated Query (LINQ)*
*Asynchronous programming*
Conclusion
Suggested readings

**5. Building Apps with Xamarin and Xamarin.Forms**
Introduction
Structure
Objective
Understanding Xamarin.Android Projects
*Understanding the project structure*
*Drawing and handling the user interface*
*Configuring the App Manifest*
*Debugging an Android app*
*Configuring debugging and Build options*
Understanding Xamarin.iOS projects
*Understanding the project structure*
*Handling the user interface*
*Debugging an iOS app*
*Understanding provisioning profiles and developer accounts*
*Configuring App Package options*
Cross-platform projects with Xamarin.Forms
*Understanding the project structure*
*Introducing XAML, App.xaml, and MainPage.xaml*

*Running and debugging apps*
Preparing apps for publication
*Preparing Android packages*
*Preparing iOS packages*
Conclusion
Points to remember
Key terms

**6. Organizing the User Interface with Layouts**
Introduction
Structure
Objectives
Understanding the concept of layout
*Alignment and spacing options*
*Understanding the visual tree*
*.NET objects hierarchy*
Organizing the user interface
*Creating a Sample Project*
*The StackLayout*
*The FlexLayout*
*The Grid*
*Spacing, proportions and spans for rows and columns*
*The AbsoluteLayout*
*The RelativeLayout*
*The ScrollView*
*Controlling the ScrollView programmatically*
*The Frame*
*The ContentView*
*Using a ContentView*
Styling the user interface with cascading style sheets
*Defining CSS styles as a XAML resource*
*Consuming CSS files in XAML*
*Creating and implementing CSS styles in C#*
Conclusion
Points to remember
Key terms

**7. Understanding Common Views**
- Introduction
- Structure
- Objectives
- Creating a sample project
  - *Common properties*
- Working with text
  - *Displaying text with the label*
  - *Accepting user input with entry and editor*
    - *Entering passwords*
  - *Applying and managing fonts*
    - *Implementing custom fonts*
    - *Complex text formatting with FormattedString*
- User interaction with buttons
- Selecting dates and time
  - *Selecting dates with the DatePicker*
  - *Selecting a time with the TimePicker*
- Displaying HTML content
- Selecting Boolean and numerical values
  - *Turning options on and off with the switch*
  - *User choices with the CheckBox*
  - *Multiple choices with RadioButton*
    - *Implementing multiple groups*
  - *Value selection with the Slider*
  - *Incremental value selection with the Stepper*
- Implementing search functionalities
- Handling long-running tasks
- Displaying images
  - *Handling the aspect*
  - *Managing image files*
- Adding interactivity to Views
  - *Implementing GestureRecognizers*
  - *Displaying and handling alerts*
- Understanding visual states
- Conclusion
- Key terms
- Suggested readings

**8. Pages and Navigation**
Introduction
Structure
Objectives
Introducing available pages
*Individual pages: The ContentPage*
*Master-details views: The FlyoutPage*
*Organizing contents within tabs: The TabbedPage*
*Scrolling pages: The CarouselPage*
Navigating between pages
*Sharing data between pages*
*Implementing custom titles*
*Understanding pages lifecycle*
*Intercepting the physical back button*
Common app features: The Shell
*Understanding the structure of the Shell*
*Adding a Flyout menu*
*Leveraging built-in navigation: The Tab bar*
*Implementing both the Flyout and Tab bar*
*Implementing the Search bar*
*Programmatically interacting with the Shell*
*Changing the Shell styles*
Conclusions
Key terms
Suggested readings

**9. Resources and Data Binding**
Introduction
Structure
Objectives
Creating a sample project
Understanding and Defining Resources
*Defining resources*
*Defining and assigning styles*
*Implementing style inheritance*
*Implementing implicit styling*
Binding data to the user interface

*Getting started with data binding*
*Property change notifications: INotifyPropertyChanged*
*Assigning the binding context*
*Working with data collections*
*Displaying data: The ListView*
*The DataTemplate as a resource*
*Binding different types: Value converters*
*Displaying Collections efficiently*
*Displaying lists with the CollectionView*
*Scrolling lists with the CarouselView*
*Displaying item indicators with the IndicatorView*
*Selecting items with the Picker*
*Introducing bindable layouts*
Advanced data binding: The Model-View-ViewModel pattern
*Defining the data model*
*Implementing the business logic: Commands and ViewModels*
*Exposing data*
*Defining actions with Commands*
*Designing the user interface*
*MVVM frameworks*
*Pull-to-Refresh gesture: The RefreshView*
Local data access with SQLite databases
*Installing the SQLite NuGet package*
*Getting the database path*
*Implementing a data model*
*Implementing a data access layer*
*Invoking the data access layer*
*Extending the user interface*
Conclusion
Key terms

**10. Brushes, Shapes, and Media**
Introduction
Structure
Objectives
Creating a sample project
Coloring objects with brushes

*Defining linear gradients*
*Defining circular gradients*
Drawing shapes
*Drawing circles and ellipses*
*Drawing rectangles*
*Drawing lines*
*Drawing polygons*
*Drawing custom shapes*
*Further studies: Path and geometries*
Working with multimedia
*Installing the Xamarin Community Toolkit*
*Implementing the MediaElement*
*Controlling the media file*
*Playing local files*
Conclusion
Key terms
Suggested readings

**11. Managing the Application Lifecycle**
Introduction
Structure
Objectives
Creating a sample project
The Application class
*Working with themes*
*Defining global variables*
Events of the Application lifecycle
*Understanding and using Application events*
*A real-world example: Storing and retrieving data*
*A real-world scenario: Restoring data forms*
*Responding to page events*
Sending messages through the app
*Broadcast messages with MVVM*
*MessagingCenter tips and tricks*
Conclusion
Key terms
Suggested readings

**12. Working with Web API**
- Introduction
- Structure
- Objectives
- Chapter prerequisites
  - *Getting a Free Azure subscription*
  - *Downloading Postman*
- Understanding web services and Web API
- JSON: A standard data exchange format
- Creating Web API in Visual Studio
  - *Understanding the project structure*
  - *Creating a data model*
  - *Implementing controllers*
  - *Retrieving data*
  - *Creating and updating data objects*
  - *Deleting data objects*
  - *Testing Web API services*
    - *Making API calls with Postman*
- Publishing Web API services
- Consuming Web API with Xamarin.Forms
  - *Creating a data model*
  - *Creating the ViewModel*
    - *Calling Web API services from C#*
  - *Designing the user interface*
  - *Testing the application*
- Conclusion
- Key terms
- Suggested readings

**13. Working with Native API**
- Introduction
- Structure
- Objectives
- Preparing a sample project
- Working with the device class
  - *Working with Timers*
  - *Running thread-safe code*

*Device-based content orientation*
*Conditional XAML: OnPlatform and OnIdiom*
Advanced view customization: Custom renderers
*Defining a custom view*
*Defining the Android renderer*
*Defining the iOS renderer*
Applying custom renderers to views
*More information on custom renderers*
Managing native properties with effects
*Declaring effects*
*Implementing platform-specific effects*
*Testing the code*
Displaying native views
Customizing views with platform-specifics
Accessing device features: The DependencyService class
Cross-platform access to native API: Xamarin.Essentials
*Checking the network connection*
*Checking the battery status*
*Sending emails and SMS messages*
*Sending emails*
*Sending SMS messages*
*Opening contents*
*Opening the web browser*
*Opening default apps*
*Storing user preferences*
*Storing secure settings*
*Secure settings with app version tracking*
*More essentials API*
*Hints about plugins*
Conclusion
Key terms
Suggested readings

**14. Finding a Job**
Introduction
Structure
Objectives

Preparing your resume
Finding jobs, the modern way: Using LinkedIn
*Searching for a job*
A step forward: Attracting jobs
*Connecting with the right contacts*
*Sharing contents from others*
*Sharing your knowledge*
*Writing articles and other content*
*Joining online communities*
*Sharing your code*
*Final considerations about LinkedIn*
Preparing for job interviews
*Technical interviews*
*Personal interviews*
Conclusion
Points to remember

**15. Succeeding as a Mobile App Developer**
Introduction
Structure
Objectives
Developing your passion and curiosity
*Experiment on custom projects*
*Experimenting with third-party components*
Learning to be a team player
Staying up-to-date with development technologies
*Attending conferences and meetups*
Conclusion
Points to remember
Suggested readings

**Index**

# CHAPTER 1

# The Importance of Mobile App Development

## Introduction

As a Xamarin jobseeker, you are preparing to enter the fantastic world of developing and publishing applications for mobile devices. Writing software certainly requires a technical background, but it also requires understanding the market and the user base you are going to target. In the case of mobile app development, it is necessary to understand why it represents an important career opportunity today and what the recommended approach to obtain the best results is. This chapter provides an introduction to the state of mobile app development today and explains why using a development framework like Xamarin can open up more opportunities, giving you hints on how to become autonomous in keeping up to date with new releases in the mobile world.

## Structure

In this chapter, we will cover the following topics:

- The importance of mobile app development today
- The importance of developing mobile apps with a cross-platform approach
- Staying up to date with technology releases

## Objectives

After completing this chapter, you will understand why building mobile apps is an important opportunity for jobseekers today, and you will also understand why it’s important to develop applications for multiple platforms from one codebase. You will get clarifications about which types of mobile devices are available today and how they are changing people lives. Finally, you will be provided with tips on how to stay up to date with new releases and platform updates, which is a crucial part of working in information technology.

## The importance of mobile app development

In the last 10 years, the incredible diffusion of smartphones, tablets, and wearable devices has literally changed our lives. If you consider smartphones, the most relevant mobile devices, they not only allow for making phone calls but also work as personal assistants for everybody. They allow people to do things that one could only do with a computer until a few years ago, like surfing the internet, or using dedicated tools and resources. Think of MP3 players to play music, clocks to see what time it is, a calendar to mark appointments, newspapers to stay informed and so on; all of this is now available in a smartphone. For you as a software developer, the increasing diffusion of mobile devices, their continuous technological evolution, and the variety of applications they can have make for an incredible opportunity for finding a long-term job in the software industry. But before you get started with development, some considerations need to be made about the evolution of mobile devices and how they changed not only our lives but also the way companies manage their work.

## Mobile devices and applications

The first point to clarify is about what a mobile device is and how this definition fits nowadays, including a bit of history that will help you focus on the importance of mobile app development today. Any device that can work in mobility could be considered a mobile device. Old cellular phones that only allowed for phone calls and text messages, laptops, watches, MP3 players, compasses, are all examples of mobile devices. Obviously, you cannot build applications for a cellular phone from the late 90s or for an MP3 player, but they are still mobile devices. So, here comes the first fundamental consideration: the modern era of mobile devices has started with the combination of advanced hardware with an operating system. Having an operating system on a mobile device, such as iOS or Android, not only brought to the end user an improved, human-centric experience in using all the built-in system features, including the camera and sensors, but also the option to have applications. This has made the real difference between the old devices and modern devices. Having an operating system on a device allows for building applications that can have different purposes and leveraging all the system hardware, opening a new world of business opportunities for individual developers, software companies, and internal enterprise usage.

## A brief history of modern mobile devices

Several operating systems are available in the market, but most mobile devices run on iOS and Android, so we might tend to think that these have been the first systems to start the mobile revolution. This is only partially true. In the middle of the 2000s, Microsoft released the Windows **Compact Edition** (**CE**) operating system, which supported the first generation of smartphones and tablets. At that time, devices running Windows CE already offered touch screens, even though it was necessary to use a plastic pen to interact with the device because interaction with the screen was basic back then. Windows CE was a fully featured operating system and included the Microsoft .NET Compact Framework platform that developers working with Microsoft Visual Studio and the C# programming language could use to build applications for that kind of devices. Windows CE was a visionary operating system but was penalized by two major issues: it was primarily thought for business customers and not for a global audience, and it was not pushed enough into people’s houses, like it happened for Windows PCs instead. Phones running Windows CE were also available in the market for everyone, but their diffusion was quite low, resulting in limited success. A few years later, Apple and Google completed the transition to the era of modern mobile devices, introducing smartphones and tablets for a global audience, with operating systems designed for a human-centric approach and not for a business-centric approach. This made it possible for every person in the world to develop applications and possibly make money from it.

## Mobile devices for everyone

Apple first introduced the iPhone with the iOS operating system in 2008, and 2 years later, they launched the iPad tablet, still running iOS. Despite the highest cost in the market, Apple put in place an incredible marketing campaign that allowed the company to push these devices, and now a lot of people often refer to the iPad as the tablet and to the iPhone as “the phone". In 2008, Google announced the first version of Android, and the first device running this operating system was an HTC Dream smartphone. The Android operating system was then quickly adopted by many producers, and it became the system of choice for a large number of smartphones and tablets at an affordable price, which has been one of the keys to their success. When releasing tablets, both Apple and Google made sure that developers could reuse most of their code to make their apps run on both device types at no cost, and this was another point in their success because end users could have their favorite apps on both the smartphone and the tablet, with only a few exceptions. Microsoft, after being the first company investing in mobile operating systems and mobile application development, joined this market again only years later with the Windows Phone operating system, first partnering with important producers like LG and Nokia and then building their own devices. Windows Phone was a very good system, but it was too late to conquer enough market share to survive, so it was finally dismissed in 2019. In the recent years, Huawei has also created an operating system for their devices built upon the Android engine and can be considered as the third most used operating system, after iOS and Android, thanks to the diffusion of their devices.

## Mobile devices everywhere

As technology evolved, tech companies were able to implement operating systems on other types of devices as well. Two examples are smartwatches and smart TVs, where you have dedicated versions of the Apple and Google operating systems that make it possible to have applications on watches and TV receivers. On a smartwatch, you can make phone calls, send messages, monitor your heartbeat, share your workout results, and do much more. On a smart TV, you can use the applications of your favorite streaming services to watch TV programs and movies. You will see later in the book how Xamarin allows creating applications for this kind of devices. Microsoft brought Windows 10 from the PC to the HoloLens holographic glasses and to the Xbox gaming console, so there are different types of devices with the vision of making applications available everywhere. Now that you have an overview of how things evolved over the last few years, it is important to understand how businesses have started to make mobile devices part of their daily procedures.

## The importance of mobile apps for companies

With smartphones and tablets entering people's houses, everyone could quickly understand the potential of these devices and their systems. It was no longer something only available inside companies for business purposes; instead, it was something for everyone, from entertainment to simplifying daily tasks, and so on. Companies are made of people, and people running companies who were using smart devices for personal purposes have quickly started to think about bringing smartphones and tablets into daily work to improve productivity, move things faster and save money. Let's look at a few examples:

- **Restaurants**: Years ago, the only way for waiters to take orders was to go to a customer's table with pen and paper, write down the order, and go to the kitchen; now, many restaurants have smartphones or tablets where waiters quickly enter the order, which automatically notifies the kitchen, ensuring efficiency and saving time for both the customer and the restaurant workers.
- **Hospitals**: With a tablet and specific applications, a doctor can check the conditions of patients in real-time at the bedside and prescribe the proper therapies, without going back and forth to the desk to check on a computer and then writing down everything they need.
- **Sales representatives:** They can go to customers and have everything they need on their tablet without bringing a laptop every time.
- **Stock management**: With a smartphone or tablet, it is possible to search for products while going around for the stock instead of first looking on a desktop workstation.
- **Banks**: By offering mobile home banking applications, they give a great service to customers who can operate their accounts from anywhere without waiting in line for hours, and they also free a lot of internal human resources.

There could be thousands of examples, but these are certainly enough to understand why mobile applications are important for businesses and, consequently, why mobile app development is so important today. The next section will further clarify the importance of the considerations mentioned above.

## Career opportunities with mobile app development

The continuous evolution of technology provides important career opportunities in the world of mobile app development. New powerful hardware, new types of devices receiving an operating system, continuous updates to mobile OS, and more companies adopting smartphones and tablets as tools for their daily work represent the security of jobs in the mobile application development industry. Opportunities can target two different segments: individual users and businesses. You can work on applications that are available to a global audience so that each person can install your apps or games, or you can work on applications that target companies or applications for internal usage that companies want to create for their specific requirements.

## Making money with mobile app development

Making money in the first segment, i.e., individual users, is certainly more difficult. There are millions of apps in the Apple Store and Google Play, and there are free apps for almost every need. Therefore, you need to invent something new and create apps or games that thousands of users are going to love so much that they are willing to pay. Building successful paid applications for a global audience is something complex and structured, so they are usually developed by software companies that can count on several teams: the development team, the marketing team, the localization team, testers, and the data analysis team.

For individual freelance developers, this can certainly be done, but the effort is very high. So, if you would like to see the result of your work running on everyone’s device, you might probably want to search for a job with a software company. As an alternative, if you wish to maintain your independence, you could still work as a freelance consultant joining one or more projects run by a software company. When it comes to companies adopting mobile devices as tools for their daily work, this is probably the most interesting option, and one that allows you to get a salary more easily. In fact, there are companies whose business is building software and companies whose business is not building software but that want to have applications for internal enterprise usage. In both cases, you will be able to work on developing new applications from zero or updating and maintaining the existing applications on a long-term basis because business requirements change over time. Moreover, the variety of business types will make your work even more interesting and challenging, which is extremely important for your motivation.

## Summarizing how a mobile system works

Previously, you have learned that the difference between modern mobile devices and old-generation mobile devices is in the fact that modern devices have an operating system that allows for hosting applications. Let's clarify these sentences in more details. Operating systems are software that make interaction possible between the user and the hardware, and they have been existing since the first computers were ever released in a global market. As an example, MS-DOS was the operating system from Microsoft empowering personal computers in the 80s. At that time, everything had to be done via the keyboard and from the command line, but the simple operation of writing a file to disk was possible, thanks to the operating system, which connected the user with the disk drive. Operating systems with a graphical user interface have then started to change the way users could use computers in the 90s, with continuous evolution that led to the operating systems we have today on our computers and devices. With regard to mobile devices, think of the system camera application you have on your phone:

- You tap the camera icon
- The software starts the camera
- With a simple button, you can control the camera to take pictures or record videos

Behind these few gestures, there are a lot of concepts that you need to know. The camera is device hardware on your phone. The OS is the real connection between you as the user and the hardware because it allows you to interact with the phone camera via the camera application. The reason why I'm continuously mentioning that the camera app is an application and not a function of the operating system is because any application built by a developer could use the camera hardware. Device producers give users a built-in system application so that the camera can be used immediately to take pictures and record videos. But if you think of messaging apps, they allow you to take pictures and quickly send them to your contacts via a message outside of the system application. This is possible because the operating system exposes functions and libraries that any application built by anyone can call to access hardware on the device. These functions and libraries are referred to as **Application Programming Interface** (**API**). Every operating system has a specific set of APIs and they are the real key to build applications for mobile devices. Camera applications pre-installed on iOS and Android invoke the system API themselves to access the hardware. APIs exist for everything that makes some work on the device, such as reading and writing files, interacting with sensors, drawing the user interface. Apple and Google provide development tools and environments that let developers access the system APIs through a programming language (like Swift for Apple and Java for Google) and a set of libraries that simplify accessing such APIs with a high-level approach. Obviously, it is not possible to describe in this book how the Apple and Google APIs work (except where expressly required in *Chapter 13, Working With Native API*), but the key point to understand here is how an operating system makes it possible for applications to use the device hardware and features.

## The importance of a cross-platform approach

Based on the discussions of the previous sections, suppose you run a company that wants to build an application to improve the business for the two major systems for mobile: iOS and Android. Apple and Google provide their own development tools, programming languages, platforms, and environments, and both the operating systems have their own specific features. With this in mind, you would have two alternatives:

- Hiring several developers, some with experience on the Apple platforms and some with experience on the Android platforms would allow your company to build an app for both OSes in a shorter time, but the cost would be much.
- Hiring less developers with experience on both development environments. This would allow for saving money, but it would at least double the development time, having a huge impact on the release timing and therefore, on your business.

Because of the demand for companies to make applications available on different platforms, during the last few years, several technologies have been created to make it possible for developers to target multiple platforms from a single codebase. This is referred to as **cross-platform development**, and it is not something that only applies to mobile development. In fact, there are technologies that allow building apps for different systems and different devices from a single, shared codebase. For instance, the Xamarin.Forms platform offered by Xamarin allows you to target Android and iOS mobile systems, Windows and macOS computers, and wearable devices from one codebase, representing a perfect example of cross-platform development technology. *Table 1.1* provides a summary of the most popular cross-platform development technologies available today:

| Name | Short description | Programming language |
|---|---|---|
| **Angular** | Angular is an open-source framework that allows for creating apps for mobile devices, desktop, and the web from a shared Java codebase. | **Java** |
| **React Native** | React Native is open-source and has been developed by Facebook. It allows for creating apps for Android, iOS, macOS, and Windows using JavaScript and the React.js library. | **JavaScript** |
| **Xamarin** | Xamarin is an open-source framework built upon the Microsoft .NET technology. It allows for creating apps for Android, iOS, macOS, Windows, and wearable devices, and it has been thought for developers with existing skills about the Microsoft stack. | **C#** |
| **Flutter** | Flutter is an open-source framework developed by Google, which allows for building apps for mobile devices and desktop systems, including Linux. It works with the Dart programming language, still created by Google. | **Dart** |
| **Cordova** | Cordova has been one of the first cross-platform development tools ever and is now backed by Adobe. Apps are the result of combining HTML markup for the user interface and JavaScript code for operational functions. | **HTML + JavaScript** |
| **Ionic** | Ionic has been built on top of Cordova and works with JavaScript code. It has often been criticized for lower performance of the generated apps. | **JavaScript** |

***Table 1.1:*** *Most popular cross-platform development technologies*

It is important to understand that companies usually make decisions about the technology to use and the people that they need to hire based on many conditions, including the existing skills of the current employees. This is one of the key points of *Chapter 2, Xamarin and Microsoft in the Mobile App Market*. This book focuses on Xamarin, but there are a few points that all these cross-platform development frameworks share. These are discussed in the following sections.

## Limitations of cross-platform code

Different operating systems have different architecture and APIs and run on different hardware. As a result, cross-platform code will not be able to access all the features that are available to a specific system; it will only be able to access the ones that multiple systems have in common. For example, an iPhone and a Samsung running Android both have a camera, and cross-platform code can certainly take pictures and record videos because this feature is available on both systems. However, the iPhone’s camera has functionalities that the Samsung device does not, and vice versa; such features will not be accessible from cross-platform code because they are specific to the platform. Continuing with the previous example, developers will still be able to separately access the full features of the iPhone’s camera and the full features of the Samsung device’s camera, but this needs to be done with native code, which means writing code that works only with a specific platform and, therefore, is not cross-platform. Practical examples about how this works with Xamarin are given in *Chapter 13, Working With Native API*.

## Application size and performance

You will often hear people talking about cross-platform applications. This terminology is incorrect. Applications are always native, where native means that a cross-platform development framework, like Xamarin.Forms can generate a binary file that is specific to the target system, such as a **`.ipa`** file for iOS, a **`.exe`** file for Windows, and a **`.apk`** file for Android. No common file format can target all the three. The correct terminology is cross-platform project instead. In the upcoming chapters, you will learn that a project is a set of source code files, images, fonts, and all the resources that you need to create an app, which will be finally bundled into the application file. These clarifications are very important because a lot of developers and often, stakeholders and decision-makers think that cross-platform means poor performance and extremely big app size. This is not true; for better understanding, let’s summarize how native and cross-platform development work in a few simple words.

## Understanding native applications

A native application built with Apple or Google development tools directly accesses the iOS or Android APIs, respectively. Instead, when you build an app with a cross-platform development framework, it has libraries that first detect what kind of operating system the app is running on, and based on the system, it will invoke the appropriate iOS or Android API to draw the user interface and invoke other functionalities. In short, your app will need the framework libraries to work, and these libraries need to be bundled into the application file produced by the cross-platform development framework. As a consequence, the generated application file will be a bit bigger because it needs to include some more libraries.

## Demystifying application performance issues

Your application will need to first invoke the cross-platform framework’s libraries, so it will go into an additional overhead. However, both the application size and this additional overhead are reasonable and sometimes, not even noticeable. With regard to Xamarin, this technology has evolved over the years, with an incredible effort from Microsoft to offer the best performance possible at the framework level.

Most of the times, if an application is performing poorly, it is because the architecture has not been designed properly or because developers have not made the best decisions when writing code. Sometimes, the final application file becomes very big because it contains unused resources, image files that are too big for a mobile device, and so on. Additionally, simply updating the development tools to the latest version might automatically bring performance improvements. Having that said, you should not be scared about application size and poor performance only because you are using cross-platform development tools. The way you architect and code the application is going to make the difference. In this book, we will provide you with practical tips and examples on how to always achieve the best performance possible.

## Staying up to date with new devices and platforms

It is important for you to stay up to date with new devices and operating system updates. In addition, when new devices are available on the market or when new versions of a mobile operating system are offered as an update to existing devices, producers also release updated developer tools that target the new offerings. This is extremely important for you because you need to ensure that your existing code still works with new devices and updated systems. Thus, it is important for you to know when updated developer tools are available and make the necessary adjustments to your work.

**In *Chapter 5, Building Mobile Apps With Xamarin and Xamarin.Forms*, I will provide specific information on how to update the development tools you need for this book.**

You are free to choose your favorite way for staying up to date, but the simplest way is bookmarking the developer websites of each producer and subscribing to news and events, also through newsletters. Especially when new devices are going to be available on the market, producers organize conferences that can be usually attended online, which you should watch. The following paragraphs provide information about subscribing for news related to the platforms discussed in this book from the major producers.

## Getting updates from Apple

Apple shares news and events' information, such as conferences where the launch of new products is announced, on their website (**https://www.apple.com**). If you're willing to work in the mobile app development industry, you might also want to bookmark the Apple Developer website (**https://developer.apple.com**). This is the main developer portal where you can access all the development resources you need to build apps for macOS, iOS, tvOS and WatchOS. These will be recalled when appropriate in the book, but here, they also share news about technical conferences, events, new devices, and new version of the operating systems from a development point of view. *Figure 1.1* shows an example of the Apple Developer portal with the announcement of the Worldwide Developer Conference 2021 (WWDC21) and a shortcut to a list of the announcements made during the event (Apple Event):

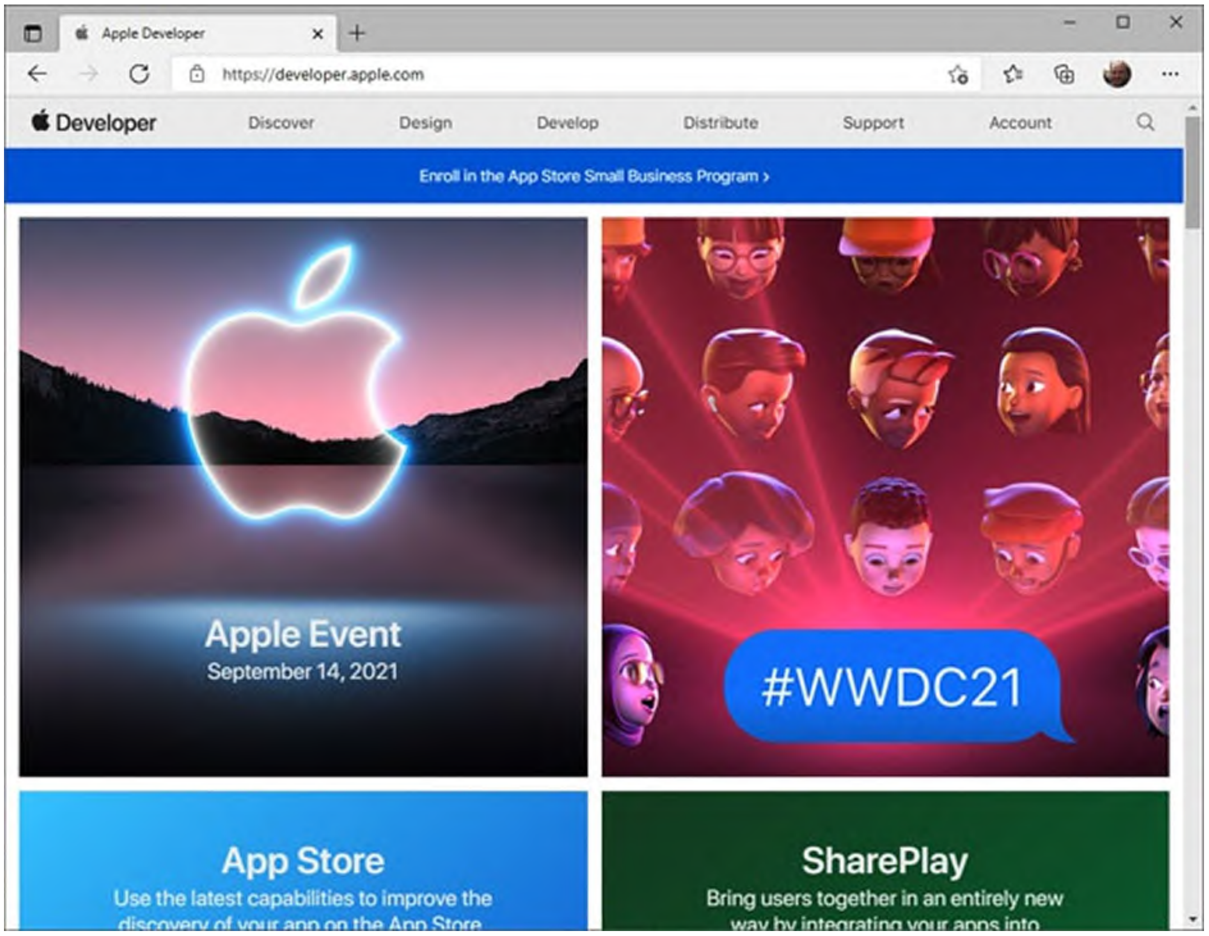
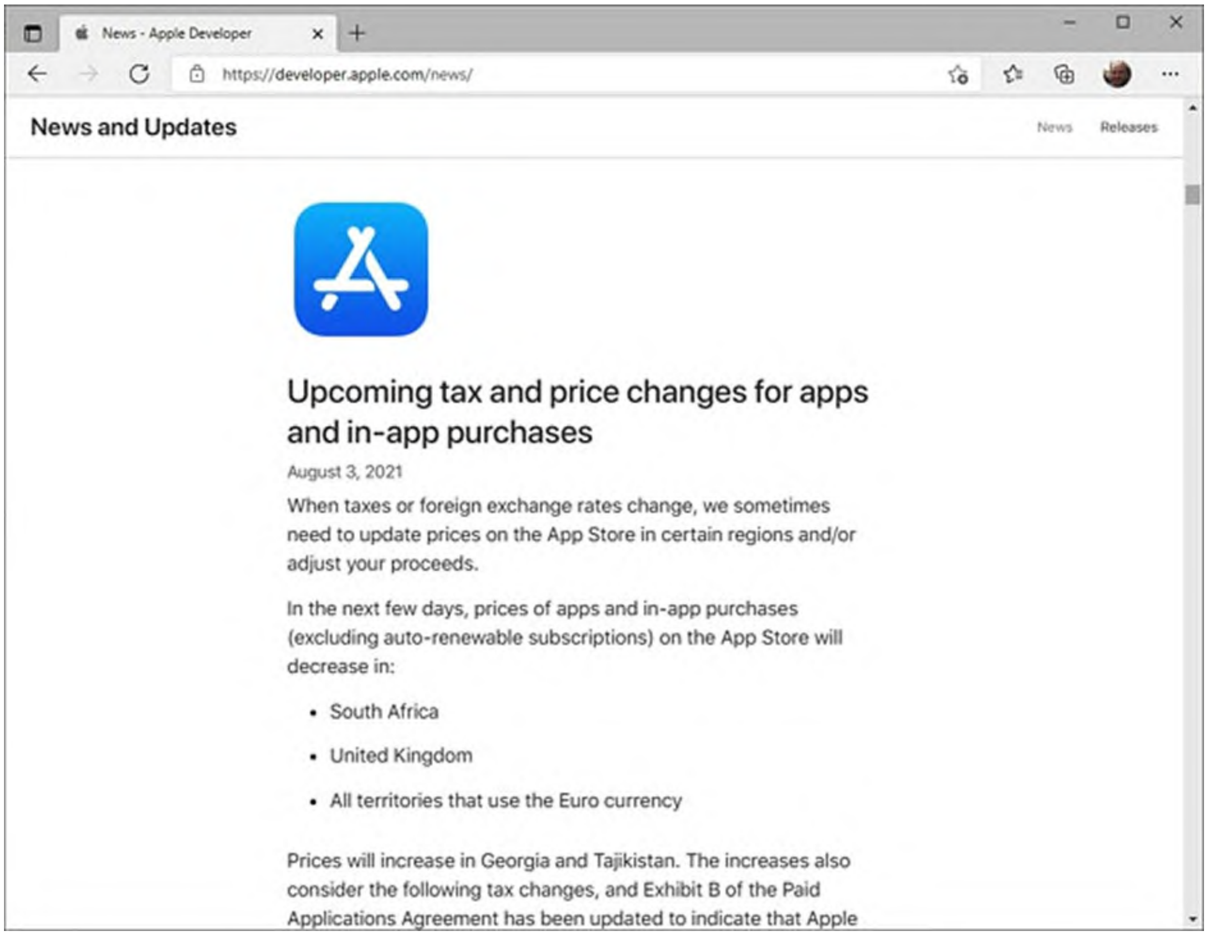

Apple Developer
https://developer.apple.com
Developer
Discover
Design
Develop
Distribute
Support
Account
Enroll in the App Store Small Business Program ›
Apple Event
September 14, 2021
#WWDC21
App Store
Use the latest capabilities to improve the discovery of your app on the App Store
SharePlay
Bring users together in an entirely new way by integrating your apps into

***Figure 1.1:*** *Getting updates from the Apple Developer portal*

You can click on each shortcut and see the list of announcements, news, new devices, and technical sessions for developers using proprietary Apple technologies. For a comprehensive list of updates that are related to submitting apps to the App Store and to new development libraries, you can visit the **`News and Updates`** website. *Figure 1.2* shows an example where you can see how the website is informing about tax changes on the App Store:



News and Updates

Upcoming tax and price changes for apps and in-app purchases

August 3, 2021

When taxes or foreign exchange rates change, we sometimes need to update prices on the App Store in certain regions and/or adjust your proceeds.

In the next few days, prices of apps and in-app purchases (excluding auto-renewable subscriptions) on the App Store will decrease in:

- South Africa
- United Kingdom
- All territories that use the Euro currency

Prices will increase in Georgia and Tajikistan. The increases also consider the following tax changes, and Exhibit B of the Paid Applications Agreement has been updated to indicate that Apple

***Figure 1.2:*** *The News and Updates web page shows updates for developers*

This website is very important because it lists mandatory requirements that you need to address when publishing apps to the App Store, and it also lists new and updated developer tools, and updates about the Store policies that you must be aware of.

## Getting updates from Google

Google shares information about the Android operating system and the development tools on two different websites. News about the Android operating system and general announcements about Android can be found at **https://www.android.com**. Scrolling the page from top to bottom, you will find shortcuts to the list of new features, design updates, support for specific technologies, and a list of devices running Android. At the bottom of the page, you will find a list of news for developers, which you can scroll horizontally. *Figure 1.3* shows how it looks:

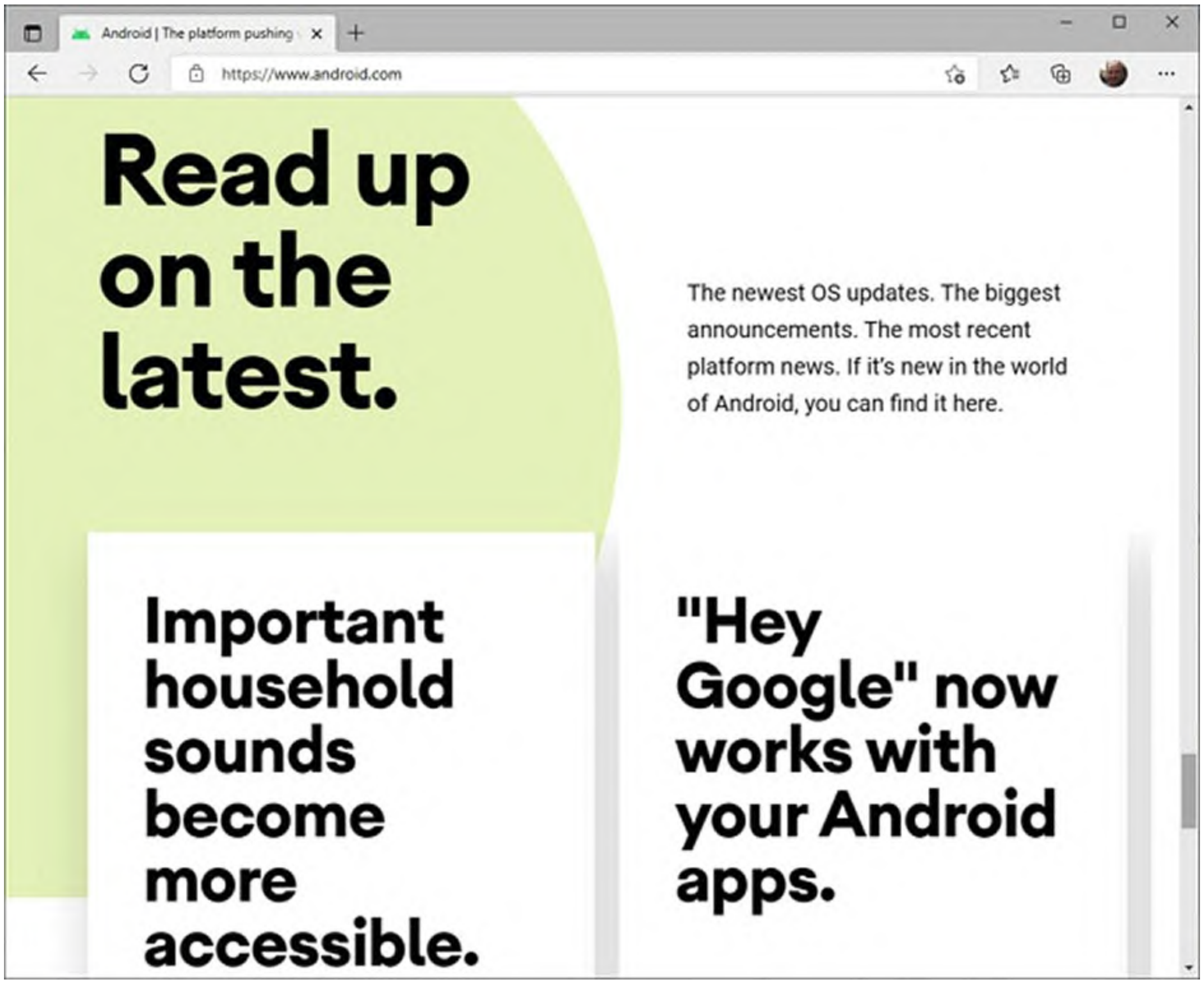

Android | The platform pushing
https://www.android.com
Read up on the latest.
The newest OS updates. The biggest announcements. The most recent platform news. If it's new in the world of Android, you can find it here.
Important household sounds become more accessible.
"Hey Google" now works with your Android apps.

***Figure 1.3:*** *Latest news about Android for developers*

The second website from which you can get news about developing for Android with Google tools is the Google Developers website (**https://developers.google.com**). This is the main developer portal and includes the official documentation for all the Google technologies supporting the development for Android. It is recommended that you visit the **`Events`** page, where you can find the list of conferences and events that you can attend to stay up to date about what's up and coming for developing apps for Android and new devices supporting this operating system.

## Getting news from Microsoft

Even if Microsoft is no longer producing smartphones, tablets (except for convertible Surface machines), and mobile operating systems, the development environment, programming language, and platforms described in this book and that you will use in your working life are produced by Microsoft. In addition, Xamarin can target Windows 10, which has a store (the Windows Store) from which users can download applications, so it is important to stay up to date with new releases from this company. Global news is available at **https://news.microsoft.com**. Events and conferences can be accessed from the Events page (**https://www.microsoft.com/en-us/events**), where you can find a list of upcoming and passed events with recordings for developers, IT professionals and businesses. Note that, even if the address of the page includes the en-US localization, this is a global page and summarizes events available worldwide. You can then browse the Microsoft Developer (**https://developer.microsoft.com**) website for the latest news, releases, and updates about development tools and platform. This is particularly important to stay up to date with new technologies and official documentation.

## Conclusion

With the increasing diffusion of the most modern mobile devices, and due to the way companies' businesses have changed over time, becoming a mobile application developer is certainly one of the best choices you can do to start a successful and secure career. This not only involves technical skills, but also understanding how the market is evolving and how users' and companies' needs change, along with the availability of new devices and new operating system versions. Smartphones and tablets are certainly the most used devices for both personal and business purposes, so you might want to focus your attention on the iOS and Android operating systems. In order to build applications, you need development tools and platforms, either native or cross-platform. There are several options available, and this book discusses Xamarin and its Xamarin.Forms flavor, which helps you target multiple systems from one, shared C# codebase in a cross-platform approach. Cross-platform development has pros and cons, but you will be able to get the most out of it if the architecture of your application is accurate. Development tools and platforms also evolve along with devices and operating systems, so you must stay up to date with announcements and releases from the major producers; you have seen how to accomplish this in a simple way. After this general overview of the importance and state of mobile app development today, it is time to go specific and understand why Xamarin is a good choice and how it fits in the market today.

## Points to remember

- **Mobile device**: Any device that can work in mobility.
- **Operating system**: Software that allows interaction between user and hardware.
- **Application Programming Interface (API)**: Libraries of the operating system that developers can consume in their applications.
- **Cross-platform development**: Creating software for multiple platforms and systems from one codebase.

# CHAPTER 2

# Xamarin and Microsoft in the Mobile App Market

## Introduction

In the first chapter, you got an overview of the status of the mobile app market today and about the most popular development frameworks for building applications for mobile devices. This chapter provides further details on Xamarin, the subject of this book, providing information about its history and its growth since Microsoft acquired it and helping you understand what you need to develop apps with this technology. You will also learn the basics of what's in the Xamarin development platform and why it can be a great choice to be successful today.

## Structure

In this chapter, we will cover the following topics:

- Xamarin as the app development framework
- Understanding Xamarin and Xamarin.Forms
- Installing and configuring the development tools

## Objectives

By completing this chapter, you will clearly understand the benefits of choosing Xamarin as the development technology to build applications for mobile devices, more specifically, its Xamarin.Forms flavor. You will learn what tools you need to use and how to install and configure the development environment, and you will also get some hints about what Microsoft is working on for the next generation of mobile app development tools.

## Xamarin as the app development framework

For a company, choosing the right development framework for building any kind of applications is crucial. It has an impact on the budget, timing, and credibility in front of stakeholders, and it also affects the developers that will work on a project. There is actually no rule, and the decision depends on several factors. However, one of the most common approaches in choosing the development framework considers the existing skills of the developers, the cost of hiring new developers if the current skills are not enough, and the cost and time to train current developers in relation to the timing of the project. Companies often realize that the best option is using a development framework that allows reusing existing skills or that requires a short training. This can potentially save a lot of time and money, and it is even more important when it comes to mobile app development, especially for companies that do not have their applications yet, making the investment a risk.

## Mobile App Development with Microsoft Skills

For companies where developers have a robust Microsoft background, such as the .NET technology and the C# programming language, choosing Xamarin as the development platform for mobile apps is a natural choice. This is the very first important point of this chapter, which is clarifying the target user base for Xamarin: developers and companies with a strong background upon Microsoft technologies. With Xamarin, you can build mobile applications using C# as the programming language, reusing all the existing skills and knowledge about the .NET technology. Additional clarifications are required to understand how you can create applications for iOS and Android with C#, so a good idea is to start with a bit of history of Xamarin.

## Xamarin as a company

Back in June 2000, Microsoft announced the .NET Framework (pronounced dot net), a modern technology that developers could leverage to build the new generation applications on Windows. In the next chapter, you will learn more about the architecture of the .NET Framework and about its evolution in the last few years that led to one, cross-platform .NET. For now, you just need to know that, among other things, .NET Framework implements the following:

- Language interoperability, a programming language in the .NET family can consume code written with other .NET languages.
- A large reusable code library that covers the most common operations.
- Automatic memory management for many scenarios.
- A safe, sandboxed environment for the execution of applications.

Today, .NET is an open-source, cross-platform, and cross-device technology, but it was all about Windows until 2017. For this reason, when the .NET Framework was first announced, Miguel de Icaza, a software engineer for the Ximian company, immediately understood its potential and started investigating whether it it could be run on Linux. Such an investigation led to the development of the Mono project, a framework compatible with .NET that could run on systems other than Windows and that allows developers to write C# code on such systems.

## Introducing the mono project

Mono (**https://www.mono-project.com**) was first released in 2004 after 3 years of development, and during the years, the development and evolution of the project focused not only on supporting desktop platforms like Linux but also mobile platforms like iOS and Android with specific derived frameworks called **MonoTouch** and **MonoDroid**, respectively. An open-source development environment called **MonoDevelop** was also created to allow developers to write C# code on top of Mono. While the development of Mono was going forward, in 2003, the Ximian company founded by *Miguel de Icaza* was acquired by *Novell*, and the project continued to evolve, pairing to the new .NET Framework releases. In 2011, *Novell* was acquired by Attachmate, which announced hundreds of layoffs, which included *Miguel de Icaza*. So, when he had to leave, he founded a new company called Xamarin with another engineer, *Nat Friedman* who became the Chief Executive Officer.

## The transition to Xamarin

Xamarin became responsible for continuing the development and evolution of Mono, with a few but relevant changes: the MonoTouch and MonoDroid frameworks became Xamarin.iOS and Xamarin.Android, respectively, and MonoDevelop evolved into a new development environment called Xamarin Studio, which was available for Mac computers as well. Xamarin.iOS and Xamarin.Android have been the first frameworks that allowed for writing native applications with C#, but they worked separately. This means that developers could leverage their existing C# and .NET skills, but they still had to create two different projects and know the API of both platforms.

**Mono is also important for another reason: it is the runtime system that makes it possible for applications created with Xamarin to run on different systems. Like the .NET Framework is required to run Windows applications written, for example, with WPF, Mono is required for iOS and Android apps created with Xamarin.**

Xamarin continued the evolution of their frameworks, and in 2014, they first released Xamarin.Forms, a new framework that allows to target multiple platforms (at that time, only iOS and Android) from a single, shared C# codebase. The goal of Xamarin.Forms was providing unified access to features that are common to all platforms, such as most of the user interface controls and some device features. At that time, the Xamarin.Forms codebase was limited. There was no declarative language to create the user interface, so everything was done in C#, and developers had to do native development in several situations. It was not a very productive framework, and probably many developers were discouraged by the effort they would have needed to make. However, Xamarin.Forms was the first cross-platform development framework for C#, and the interest rose quickly. A lot of improvements were made in a few months; for example, they introduced a basic version of the **eXtensible Application Markup Language** (**XAML**) to design the user interface in 2015 and several new features that brought Xamarin.Forms to be a very interesting choice. However, Xamarin licenses for enterprises were quite expensive, and despite the improvements, the adoption of this technology seemed to be stuck. However, things quickly changed, as explained in the upcoming sections.

## Microsoft in the mobile app ecosystem

Developers at Xamarin perfectly knew that most C# developers would have preferred using Microsoft Visual Studio as the development environment instead of Xamarin Studio, at least on Windows. For this reason, they created extensions for Visual Studio that made it possible to create Xamarin projects from within Microsoft's environment.

**You will often hear about extensions when talking about Visual Studio. An extension can be thought of as a plug-in that adds functionalities to the development environment. Therefore, when referring to Visual Studio, the proper terminology is extension, not plug-in.**

In addition, with such extensions, Visual Studio included visual tools that simplified the way the user interface could be designed in Xamarin.iOS and Xamarin.Android projects. So, Microsoft had a clear idea of Xamarin's potential. In fact, in February 2016, Microsoft announced the acquisition of the Xamarin company, bringing in-house all the cross-platform and native development technologies they built, hiring their engineers. This huge investment brought Microsoft back into the mobile app ecosystem, but with a different approach.

## The importance of Microsoft investments on Xamarin

As you might recall from *Chapter 1, The Importance of Mobile App Development*, Microsoft was probably not very successful as a provider of mobile OS and devices, but they have always had the best development tools and environments on the planet, so they were able to connect the power and productivity of the Visual Studio development environment with the Xamarin frameworks in order to provide the best experience to developers with existing skills on the Microsoft stack and willing to create apps for mobile devices. In the last 4 years, Microsoft has incredibly pushed Xamarin in several directions. The codebase has been improved and evolved until to have what is available today, introducing dozens of controls, reusable code, and access to device features so that the developer productivity has dramatically increased. They have open-sourced the whole Xamarin platform (along with all the technologies based on .NET), moving the source code to GitHub, which not only means accepting contributions from the community but also involving and engaging the community in the growth of the product.

## Improved productivity with Microsoft tools and services

Based on the experience gained by providing first-class tools to develop applications for Windows Phone and Windows desktop, Microsoft could extend to Xamarin and Xamarin.Forms the availability of powerful productivity tools integrated in the development environment. Not limited to this, and always keeping in mind the new cross-platform approach, Microsoft also built Visual Studio for Mac, which was first released in 2017. This was the natural evolution of Xamarin Studio, but with all the well-known power of Visual Studio, which developers could use for the first time on Mac computers. Microsoft has also created an ecosystem of products and services around Xamarin, such as application lifecycle management tools as part of Azure, Microsoft's cloud platform, services that quickly allow developers to implement push notifications, data synchronization, authorization, and other services like analytics, which help decision makers drive business and investments based on the analysis of the application usage.

By pushing Xamarin so much, Microsoft has also been able to increase trust in their mobile services. Today, with Microsoft Visual Studio 2022, developers have the most powerful development environment available to build their applications for mobile devices using Xamarin. Now that you know where Xamarin is coming from, you need to know something more about the technical aspects, which is the topic of the next section.

## Understanding Xamarin and Xamarin.Forms

Xamarin is a development technology that makes it possible for developers skilled in C# and the Microsoft stack to build applications for several operating systems. More specifically, Xamarin groups the following development platforms:

- **Xamarin.Android**, a set of .NET libraries and tools that allow you to run C# code on Android devices. This is possible because Xamarin.Android translates, behind the scenes, all the work done in C# into the Java equivalent and invokes the proper Java and Google tools to create the application binaries. As an implication, Xamarin.Android requires the Java **Software Development Kit** (**SDK**) and tools like the Android SDK Manager and the Android Device Manager. Later in this chapter, you will see how to properly configure the environment and install the necessary tools.

- **Xamarin.iOS**, a set of .NET libraries and tools that allow you to run C# code on iPhone and iPad devices. Behind the scenes, Xamarin.iOS translates all the work done in C# into the Objective C equivalent and invokes the Apple developer tools to create the application binaries. As for Android, the next section will explain how to properly set up your development environment to target iOS.
- **Xamarin.Mac**, a set of .NET libraries and tools that allow you to run C# code on macOS machines, building desktop applications. Like Xamarin.iOS, it translates all the work done into the Objective C equivalent and invokes the Apple developer tools to create the application binaries.
- **Xamarin.Forms**, one library that makes it possible to target multiple platforms from a single, shared codebase. Behind the scenes, Xamarin.Forms invokes both Xamarin.Android and Xamarin.iOS, and then these two platforms do the job of invoking the appropriate native tools to create the application binaries.

Xamarin.Android and Xamarin.iOS make it possible to access the native API of the operating systems they target, whereas Xamarin.Forms needs to pass through them. This is also why you will often hear developers talk about Xamarin native when referring to Xamarin.Android and Xamarin.iOS.

## Advantages of Xamarin.Forms

Xamarin native platforms require you to know the operating systems' API in detail. Moreover, you will still need two different projects: one for Android and one for iOS. You will be able to share some logic between the two, but all the user interface and access to the device features need separate work and effort. With Xamarin.Forms, you have the following advantages instead:

- You do not really need to know the native API in detail (though always recommended) because you have code that is common to all the platforms.
- You can target multiple platforms from one codebase.
- Xamarin.Forms can also target the Universal Windows Platform from the same codebase, allowing you to make your code run on Windows 10 as well.

- You write, debug, and maintain code once, not twice, with a reduced effort.
- You will still publish two different applications, but this is also what you would do with Xamarin.Android and Xamarin.iOS.

Visual Studio on Windows does not support Xamarin.Mac, so this will not be discussed further. In *Chapter 5, Building Mobile Apps With Xamarin and Xamarin.Forms*, you will get an overview of how creating apps with Xamarin.Android and Xamarin.iOS works, and Xamarin.Forms will be discussed more thoroughly.

## Hints about .NET MAUI

Microsoft has just released the next-generation framework for developing applications in a cross-platform approach. This framework is called .NET MAUI, which stands for Multiplatform Application User Interface, and is intended to work with .NET 6. Among the others, .NET 6 unifies the API from different technologies, like Xamarin, into one .NET codebase and replaces Mono as the runtime platform required to run mobile apps. .NET MAUI is a full cross-platform development framework that targets mobile devices as well as desktop systems. It can be considered as the natural successor and replacement of Xamarin.Forms, and you will learn more details where appropriate. Microsoft is aware that thousands of developers around the world have been using Xamarin.Forms to build applications that are critical for their business, so they have architected .NET MAUI in a way that allows keeping the same methodology of work, the same code syntax, the same tools, and the same Xamarin.Forms skills. There will certainly be some changes, but completely affordable once you know Xamarin.Forms. Moreover, the project structure will be much simpler and app performance will be much better at no cost. So, even if Xamarin.Forms is gradually replaced by .NET MAUI in the next few years, this can be an opportunity for you. In fact, as a Xamarin jobseeker, it is even more important to learn Xamarin and Xamarin.Forms today because you will be immediately ready to use .NET MAUI with the skills you acquire with this book, and you will be able to work on existing Xamarin.Forms projects that companies will not migrate soon and that still need to be maintained over time.

**Tip: Microsoft will include a Migration Assitant tool, whose goal will be**

**to simplify the migration from Xamarin.Forms projects to .NET MAUI. However, keep in mind that it usually takes time before companies decide to make such important updates to their critical applications. There are several reasons behind this, but from a technical and development point of view, you might want to consider that sometimes, it is better to wait a few weeks and see if bugs are reported and solved, and that components and libraries produced by third parties might not be available at the same time as the new technology.**

In general, you should not be too afraid of technological changes and replacements because this is part of your job in information technology. Technology evolves and new tools come up, and you, as a developer, must learn to respond to these changes. Always try to see this as an opportunity, for example, learning how to help companies migrate to a new platform before someone else can do, and before their competitors will.

## Installing and configuring the development tools

There are some prerequisites for building applications for mobile devices with Xamarin. First, if you want to build apps for iOS, you will need a Mac. This is required by the Apple policies, which state that app binaries for iOS can only be generated on macOS, regardless of the development technology used. If you do not want (or you cannot) purchase a Mac machine but still want to target iOS, you can rent a Mac online. Services like MacInCloud (**https://www.macincloud.com**) allow you to rent a Mac machine with all the development tools pre-installed so that you are ready to go. These are paid services, so make sure you choose an appropriate plan only when required.

**One limitation of a Mac device rented online is that you will not be able to test your apps on a physical iPhone or iPad, which should be connected to the Mac. However, as you will see in the next chapter, you can test the applications on device simulators.**

After these general considerations, let’s discuss the system configuration you need for both Windows and macOS in detail.

## System requirements for Xamarin

This section describes the minimum system requirements for installing the necessary development tools to build applications with Xamarin and that are also required to successfully complete all the chapters in this book. For your convenience, system requirements are listed for both Windows and macOS, but this book is based on Windows and the related configuration, so for macOS, you will only get information about what's required to produce app binaries. The reason for this is that most companies with a background on the Microsoft stack have necessarily worked with Windows for many years, so this is also the system of choice for building mobile apps.

### System requirements on Windows

The official documentation (**https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/requirements**) states that Windows 7 and Visual Studio 2017 are the minimum software requirements, and that Windows 10 is, instead, required to build apps targeting UWP. However, based on the experience of years of work with Xamarin, the recommended system requirements are the following:

- A PC with at least 8 Gigabytes of RAM memory. This will make a difference when you work with device simulators and will help keep your machine fast enough.
- Windows 10 as the operating system, not only to also target UWP from Xamarin.Forms but also because it's the most recent and therefore, powerful and optimized system.
- Microsoft Visual Studio 2019 or 2022. The latter is currently the most recent version available, and Microsoft offers a free Community edition that you will be able to use at no cost.

**The Visual Studio setup program will install all the necessary libraries and software development kits, including the Java and Google ones. This will be clarified in the next section.**

In the next section, you will walk through the installation and configuration process of Visual Studio 2022 so that you will get the environment ready.

## System requirements on macOS

On a Mac machine, all the development tools would also work with only 4 GB of RAM, but performance would not be acceptable. The following is a list of the recommended requirements:

- A Mac computer with at least 8 Gigabytes of RAM memory. Like Windows, this will help keep your machine fast enough.
- The macOS 10.14 (Mojave) or later.
- Microsoft Visual Studio for Mac 2019. At this writing, this is currently the most recent version available, but Microsoft is working on releasing version 2022 soon. A free Community edition is also available to be used at no cost.
- Xcode 10 or higher. Xcode is the development environment from Apple, which also includes the development tools that Xamarin.iOS will invoke behind the scenes. It can be downloaded for free from the App Store on your Mac.

In order to install Visual Studio 2019 for Mac on your computer, visit the Visual Studio for Mac official page (**https://visualstudio.microsoft.com/vs/mac/**) and follow the steps to download, install, and configure the development environment.

## Android and iOS Devices

Whether you work on Windows or on macOS, the development tools come with device simulators that you will be able to use to test your applications. A device simulator is a virtual machine on which a specific version of the operating system is installed and that is configured to also simulate a specific hardware configuration.

Simulators then give you the option to build apps for iOS and Android without the need of physical devices, but this is not ideal. Users, customers, stakeholders will use your apps on a physical device, and that's where you should test your code. In addition, if you need to show the result of your work to customers or even your employer, it's not professional to do so on a simulator. Obviously, you cannot buy hundreds of different devices, but one Android device and one iPhone (if you target iOS) would be enough. Then, device simulators can be a good companion to test code on different operating system versions and hardware configuration, but at least one physical device should be a requirement for you.

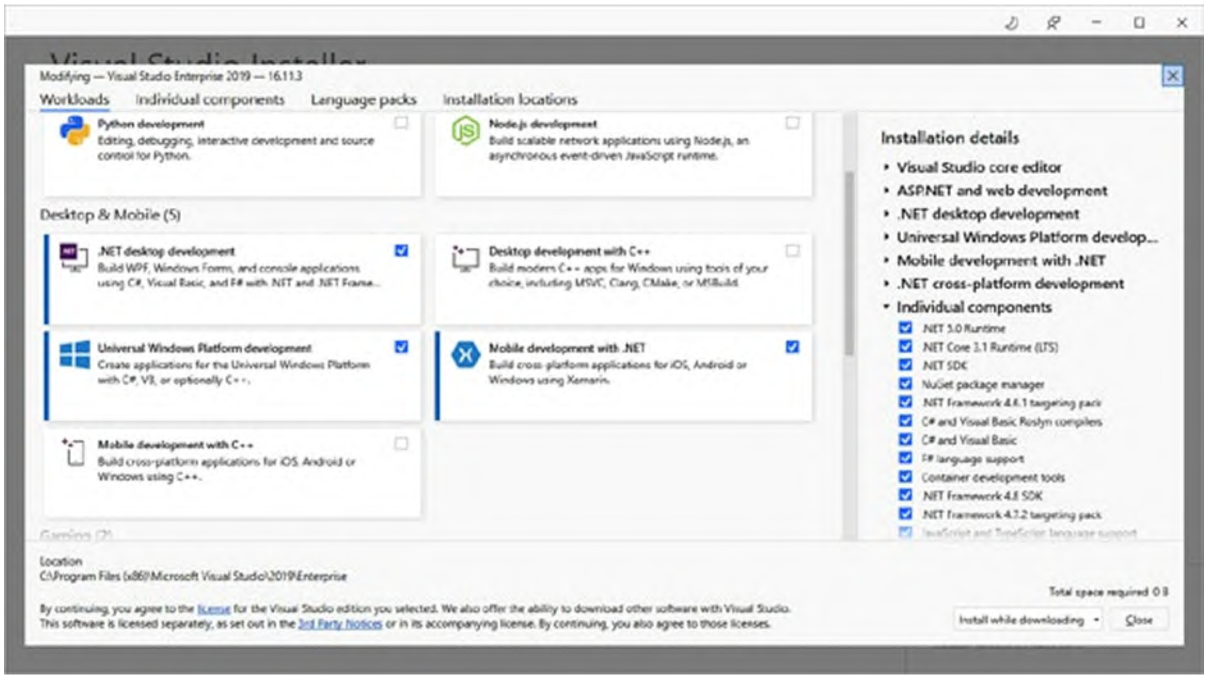
## Installing and configuring Microsoft Visual Studio

Microsoft Visual Studio is the development environment you use to build mobile apps on a Windows workstation. It is available in different editions: Enterprise, Professional, and Community. Both the Enterprise and Professional are paid editions and are usually available through appropriate licenses. If you will join a company, you will likely get one of these through a volume license. The Community edition is completely free of charge. It includes less tools and services, but it has everything you need to build mobile apps with Xamarin, and it can be used for commercial purposes.

Assuming you will be using the Community edition, visit the Visual Studio official page at **https://visualstudio.microsoft.com/vs/**. Click on the **`Download Visual Studio`** button and select the **`Community 2022`** option. This will download the Visual Studio Installer program on your disk, which you need to launch.

### Selecting workloads

When the Visual Studio Installer starts, it will offer several so-called workloads. Each workload contains the necessary tools to target a specific development area, such as desktop, web, cloud and mobile. Locate the desktop and mobile workload (see *Figure 1.2*) and make sure you select the **`.NET desktop development`**, **`Universal Windows Platform development`**, and **`Mobile development with .NET`**. The latter also includes the necessary Java and Google support tools.


Modifying — Visual Studio Enterprise 2019 — 16.11.3

[ ] **Python development**
Editing, debugging, interactive development and source control for Python.

[ ] **Node.js development**
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

[✓] **.NET desktop development**
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...

[ ] **Desktop development with C++**
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

[✓] **Universal Windows Platform development**
Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

[✓] **Mobile development with .NET**
Build cross-platform applications for iOS, Android or Windows using Xamarin.

[ ] **Mobile development with C++**
Build cross-platform applications for iOS, Android or Windows using C++.

Gaming (2)

Installation details

- Visual Studio core editor
- ASP.NET and web development
- .NET desktop development
- Universal Windows Platform develop...
- Mobile development with .NET
- .NET cross-platform development
- Individual components
  - [✓] .NET 5.0 Runtime
  - [✓] .NET Core 3.1 Runtime (LTS)
  - [✓] .NET SDK
  - [✓] NuGet package manager
  - [✓] .NET Framework 4.6.1 targeting pack
  - [✓] C# and Visual Basic Roslyn compilers
  - [✓] C# and Visual Basic
  - [✓] F# language support
  - [✓] Container development tools
  - [✓] .NET Framework 4.8 SDK
  - [✓] .NET Framework 4.7.2 targeting pack
  - [✓] JavaScript and TypeScript language support

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

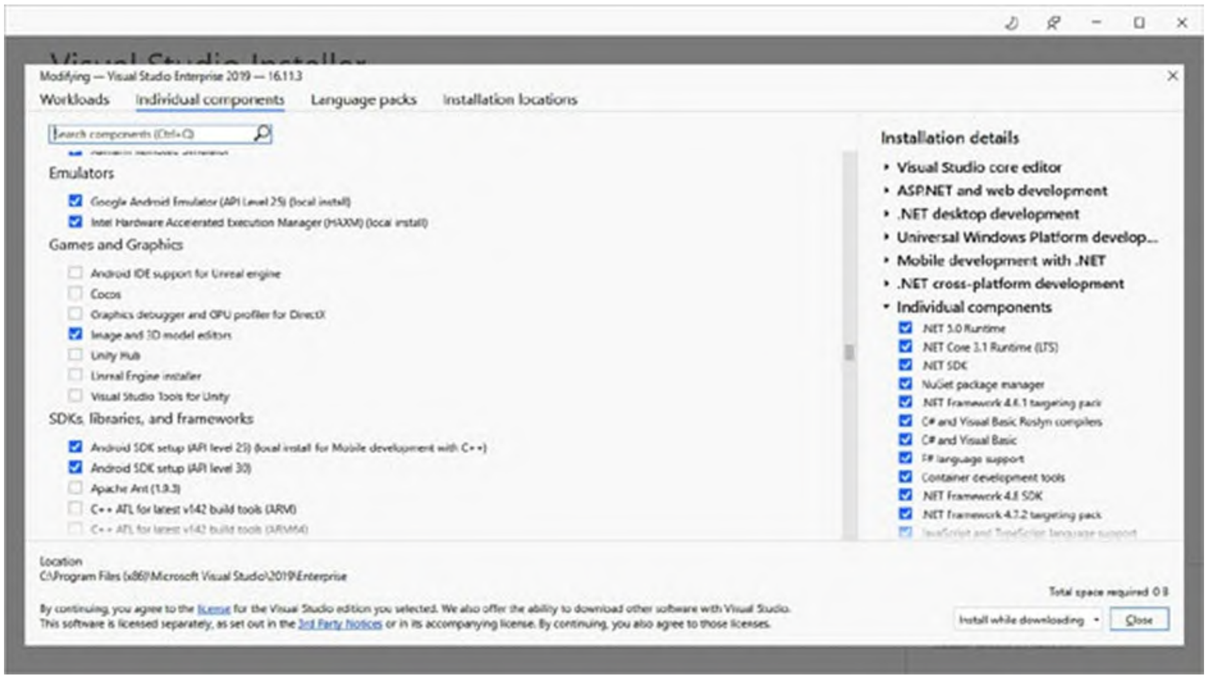
Total space required 0 B

***Figure 2.1:*** *Selecting the appropriate workloads*

**Actually, selecting Universal Windows Platform development is only required if you plan to target Windows 10 from your Xamarin.Forms projects. However, later in the book, you will see the full project structure, including the UWP project, so the workload is necessary to fully align with the contents of the book.**

## Checking for individual components

When you select the workloads, Visual Studio Installer already makes sure that necessary components will be installed, but it is important to make sure you really have everything you need. For this reason, click on the `Individual components` tab at the top of the window (see *Figure 1.2*) and then scroll down the list until you see two groups called Emulators; and SDKs, libraries, and frameworks, as shown in *figure 2.2*:


Visual Studio Installer

Modifying — Visual Studio Enterprise 2019 — 16.11.3

Workloads | Individual components | Language packs | Installation locations

Search components (Ctrl+Q)

Emulators

- [✓] Google Android Emulator (API Level 25) (local install)
- [✓] Intel Hardware Accelerated Execution Manager (HAXM) (local install)

Games and Graphics

- [ ] Android IDE support for Unreal engine
- [ ] Cocos
- [ ] Graphics debugger and GPU profiler for DirectX
- [✓] Image and 3D model editors
- [ ] Unity Hub
- [ ] Unreal Engine installer
- [ ] Visual Studio Tools for Unity

SDKs, libraries, and frameworks

- [✓] Android SDK setup (API level 25) (local install for Mobile development with C++)
- [✓] Android SDK setup (API level 30)
- [ ] Apache Ant (1.9.3)
- [ ] C++ ATL for latest v142 build tools (ARM)
- [ ] C++ ATL for latest v142 build tools (ARM64)

Installation details

- Visual Studio core editor
- ASP.NET and web development
- .NET desktop development
- Universal Windows Platform develop...
- Mobile development with .NET
- .NET cross-platform development
- Individual components
  - [✓] .NET 5.0 Runtime
  - [✓] .NET Core 3.1 Runtime (LTS)
  - [✓] .NET SDK
  - [✓] NuGet package manager
  - [✓] .NET Framework 4.6.1 targeting pack
  - [✓] C# and Visual Basic Roslyn compilers
  - [✓] C# and Visual Basic
  - [✓] F# language support
  - [✓] Container development tools
  - [✓] .NET Framework 4.8 SDK
  - [✓] .NET Framework 4.7.2 targeting pack
  - [✓] JavaScript and TypeScript language support

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise

Total space required 0 B

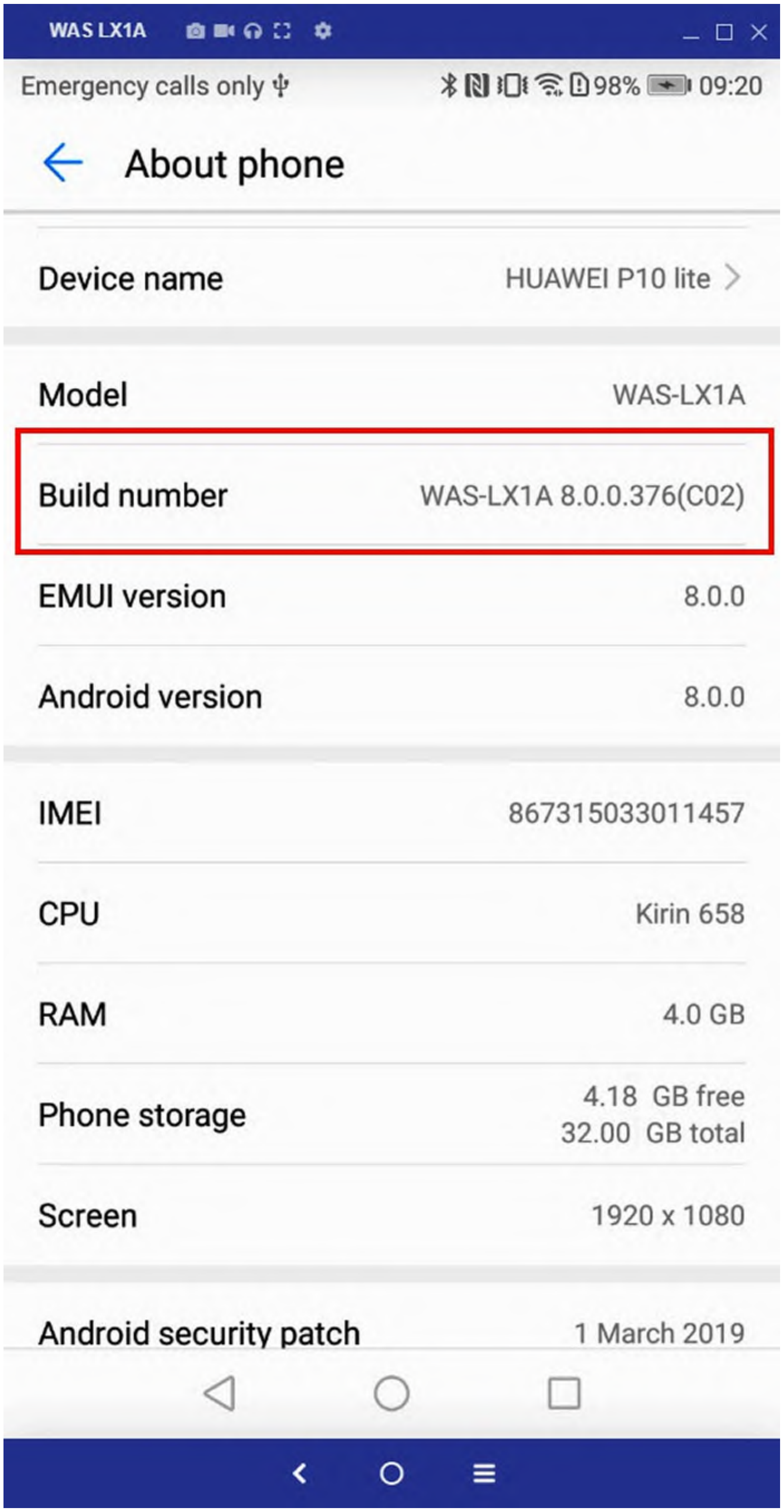
By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

***Figure 2.2:*** *Checking for individual components*

Under Emulators, make sure that both the options are selected. Under SDKs, libraries, and frameworks make sure that both the **`Android SDK setup`** options are selected. When ready, click on **`Close`**. At this point, click on the **`Install`** button and wait for the installation to complete. In the next chapter, you will start working with Visual Studio in practice.

## Configuring Android devices for developer mode

By default, Android physical devices are not enabled to support app development, so you need to enable the so-called **developer mode**. This can be done by pressing 7 times the build number of the operating system in the system information of the device. On Android, the operating system information is not in the same place always. *Figure 3.2* shows the **`Build number`** version on my Huawei P10 lite, located under the **`About phone`** menu.


WAS LX1A

Emergency calls only 98% 09:20

About phone

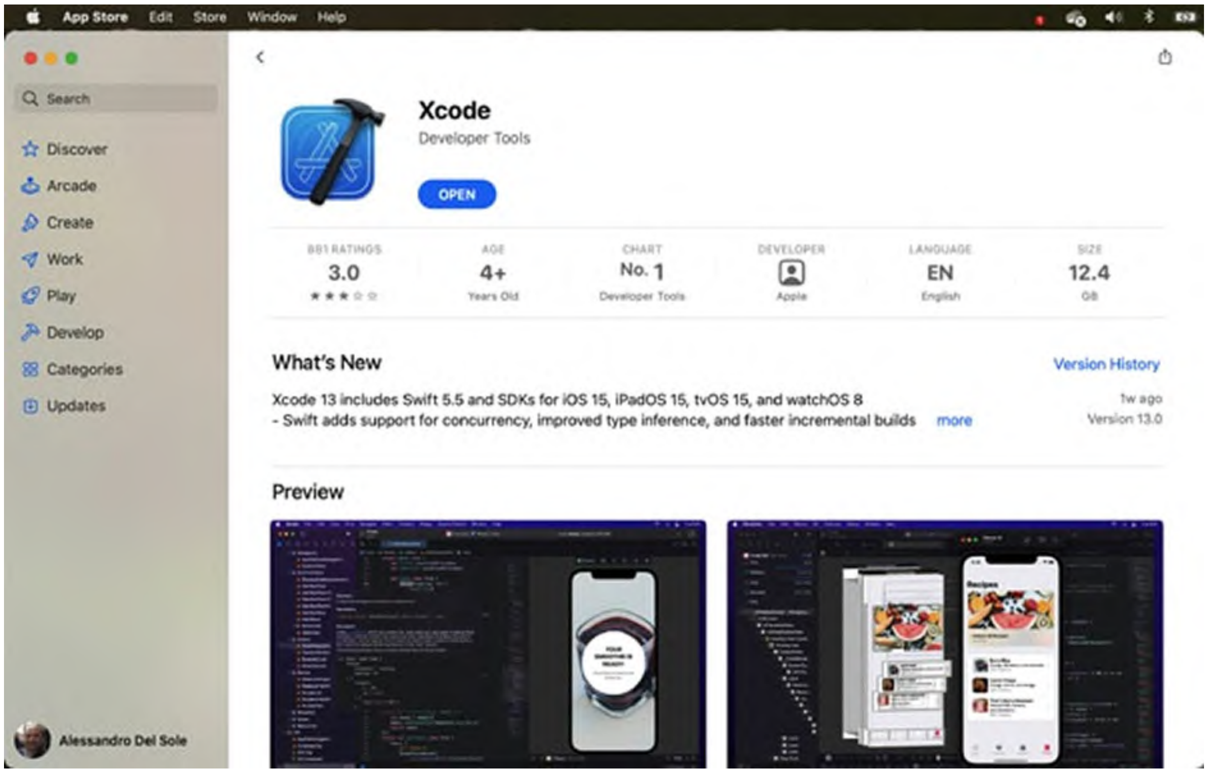
| | |
|---|---|
| Device name | HUAWEI P10 lite |
| Model | WAS-LX1A |
| Build number | WAS-LX1A 8.0.0.376(C02) |
| EMUI version | 8.0.0 |
| Android version | 8.0.0 |
| IMEI | 867315033011457 |
| CPU | Kirin 658 |
| RAM | 4.0 GB |
| Phone storage | 4.18 GB free 32.00 GB total |
| Screen | 1920 x 1080 |
| Android security patch | 1 March 2019 |

***Figure 2.3:*** *Enabling the developer mode with the OS build number*

After you tap the build number 7 times, a message will inform you that the developer mode has been enabled. This action will enable a new menu called **`Developer`** options under the system settings. Open this new menu and make sure that the USB debugging option is enabled.

## Minimum macOS configuration

As you learned at the beginning of this section, a Mac computer is required by the Apple policies to build the application binaries for iOS. If you intend to target iOS, and if you want to follow this book consistently, you need a few configuration steps on the Mac. Luckily enough, Visual Studio 2019 has simplified the way a Mac must be configured for Xamarin development, so many manual steps that were required previously are no longer necessary. On your Mac, open the App Store application and search for Xcode. *Figure 4.2* shows how this will look:



Xcode 13 includes Swift 5.5 and SDKs for iOS 15, iPadOS 15, tvOS 15, and watchOS 8
- Swift adds support for concurrency, improved type inference, and faster incremental builds

***Figure 2.4:*** *Locating and installing Xcode on a Mac*

Assuming that you do not have Xcode yet, you will see a button called

`Install` instead of `Open`. Click on the button and wait for Xcode to be installed. Note that this can take some time. Once the installation is complete, launch Xcode and wait a few minutes for its first initialization. You can then close it and wait the appropriate chapters to understand more about how it works behind the scenes with Xamarin.iOS.

**You will soon learn that, in the real world, when new updates of Xcode are available, it usually takes some days, or even weeks, to get updates of Xamarin.iOS, Xamarin.Forms, and Visual Studio that match such updates. For this reason, it is recommended to wait for the availability of updated Microsoft tools and libraries that target the new versions of Xcode before updating, otherwise you will no longer be able to run your C# code on iOS.**

## Conclusion

When it comes to choosing a development framework for mobile apps, Xamarin is a natural choice for developers with existing skills on the Microsoft stack, such as the C# programming language and Visual Studio as a development environment. The reason is that Xamarin makes it possible to write C# code and generate native iOS, Android, and UWP applications. More specifically, with Xamarin.Forms, you can write code once and target all the supported platforms from a single, shared codebase. You work with Xamarin and Xamarin.Forms from Visual Studio 2019, Microsoft's premiere development environment, and you can work on both Windows PCs and Mac computers. This chapter gave you a more detailed overview of Xamarin and helped you install Visual Studio 2022 to start development, but before you put your hands down on writing code, you need to know how the .NET technology works and the basics of Visual Studio. This is the goal of the next chapter.

## Questions

1. **What is Mono?**

   Mono is a cross-platform framework that makes it possible to write and run C# code on different systems like Linux and mobile systems.

2. **What is Xamarin.Forms?**

Xamarin.Forms is a development platform that allows for targeting multiple systems like iOS, Android, and UWP from a single, shared C# codebase.

## Points to remember

- You can use the free Community edition of Visual Studio 2019 to build apps with Xamarin even for commercial purposes.
- You can develop apps on both Windows and macOS.
- A Mac computer is required to build apps for iOS.
- The developer mode on an Android device can be enabled by tapping 7 times the operating system’s build number in the device settings.

# CHAPTER 3

# Introducing .NET and Visual Studio

## Introduction

As a developer working with Xamarin, you need to understand the technology that empowers your applications and the tools you use to write, debug, and compile code. Microsoft .NET is a technology that includes the execution environment and tools for building modern applications for the desktop, the web, mobile, and the cloud. .NET is a very powerful yet complex technology; in this chapter, you will learn the basics of its architecture. You will also get a practical introduction to Microsoft Visual Studio, the development environment you use to create Xamarin projects. In this chapter, you will also learn important concepts that are used throughout the rest of the book and get introduced to terminology that will always be important in your life as a developer.

**This chapter discusses concepts, tools, general rules, and practices that are common to all the .NET platforms and to a variety of applications. For now, it is more important for you to understand how .NET and Visual Studio work rather than getting specific information about Xamarin that would not be useful yet. The general concepts you learn here will find specific explanations starting from *Chapter 5: Building Mobile Apps with Xamarin and Xamarin.Forms*.**

## Structure

In this chapter, we will cover the following topics:

- Introducing .NET
- .NET 6: One .NET
- Building applications with Visual Studio
- Working with the Visual Studio 2019 IDE

- Compiling, running, and debugging code

## Objectives

After completing this chapter, you will know the basics of the .NET development technology, and you will have understood why it is necessary to have this knowledge before working with Xamarin and the purpose of each major component in the .NET architecture. You will also get started with Microsoft Visual Studio, the **integrated development environment** (**IDE**) that you use to write applications upon .NET. More specifically, you will learn how to create projects, debug, and run code, and how to use the most common tools and windows in the IDE.

**An integrated development environment is a software that includes all the necessary development tools in one place, like the code editor, debugger, automated tools to launch compilers, and so on. The IDE acronym will be often used in the book when referring to Visual Studio for quick reference.**

## Introducing .NET

Generally speaking, applications need an execution environment that offers services, tools, and libraries. For many years, the Microsoft .NET Framework has been the execution environment and development technology of choice for applications written with the .NET languages: C#, F# and Visual Basic. .NET is a very powerful technology, but it has two important limitations: it only works on Windows and does not support mobile development. In 2015, Microsoft announced a new platform called .NET Core, which would also run-on systems like macOS, Linux, and mobile devices. The two have been living in parallel for years but sharing all the API between the two is not always easy or feasible. For this reason, Microsoft has recently shipped .NET 6, one .NET platform that targets all the systems, from a shared codebase. Understanding the .NET Framework is extremely important for you, even if this is not really used in the book, because it is the foundation of everything about C# and .NET development, including Xamarin. In this chapter, you will start by looking at the .NET Framework and the path that first brought it to .NET Core, and finally, to .NET 6.

## Introducing the .NET framework for Windows

The .NET Framework includes two major areas: an execution environment for running applications, and tools with libraries that developers use to build applications for Windows and the Web. The latest version available is 4.8, and more about versioning is discussed later in this chapter.

As an execution environment for applications, the .NET Framework lives between Windows and your applications. As a development technology, .NET Framework 4.8 contains libraries and command line tools that are offered for free. In theory, you could write the most complex .NET application with the Windows' Notepad and compile it with the .NET tools. This is not certainly ideal, so Microsoft provides Visual Studio to build applications for the .NET Framework, a powerful environment that you meet in the second part of this chapter. C#, Visual Basic and F# are languages that developers can use to build .NET applications. A flavor of C++ can also be used to build .NET applications, but in general, this language is considered for a wider range of purposes. Regardless of the programming language you use, the .NET Framework allows you to build the following types of applications:

- Web applications with ASP.NET.
- Windows desktop applications with **Windows Presentation Foundation** (**WPF**).
- Windows desktop applications with Windows Forms.
- Network services for data exchange with **Windows Communication Foundation** (**WCF**) and Web API.

Other kinds of applications can be developed with .NET Core, as described in the next section.

**Windows Forms is still supported by Microsoft, but it is an obsolete platform and should only be used for maintaining existing programs.**

### Locating the .NET framework on disk

Usually, you will not need to directly interact with the .NET Framework tools, but sometimes it might be necessary, so it is important for you to know where to find it. On disk, .NET Framework 4.8 can be found in the

**`C:\Windows\Microsoft.NET\Framework\4.0.30319`** folder. As you can quickly verify via Windows Explorer, such a folder contains many **`.dll`** files that represent the Base Class Library and many commands line tools, such as the C# compiler and **`MSBuild.exe`**, a tool that is capable of building an entire solution. Behind the scenes, Visual Studio invokes MSBuild when you compile your code into binaries. In general, Visual Studio does all the necessary jobs for you, but it is still important to know where tools and libraries are located.

## The .NET framework architecture

The different components of the .NET Framework work as layers between Windows and your applications, as you can see in *Figure 3.1*:

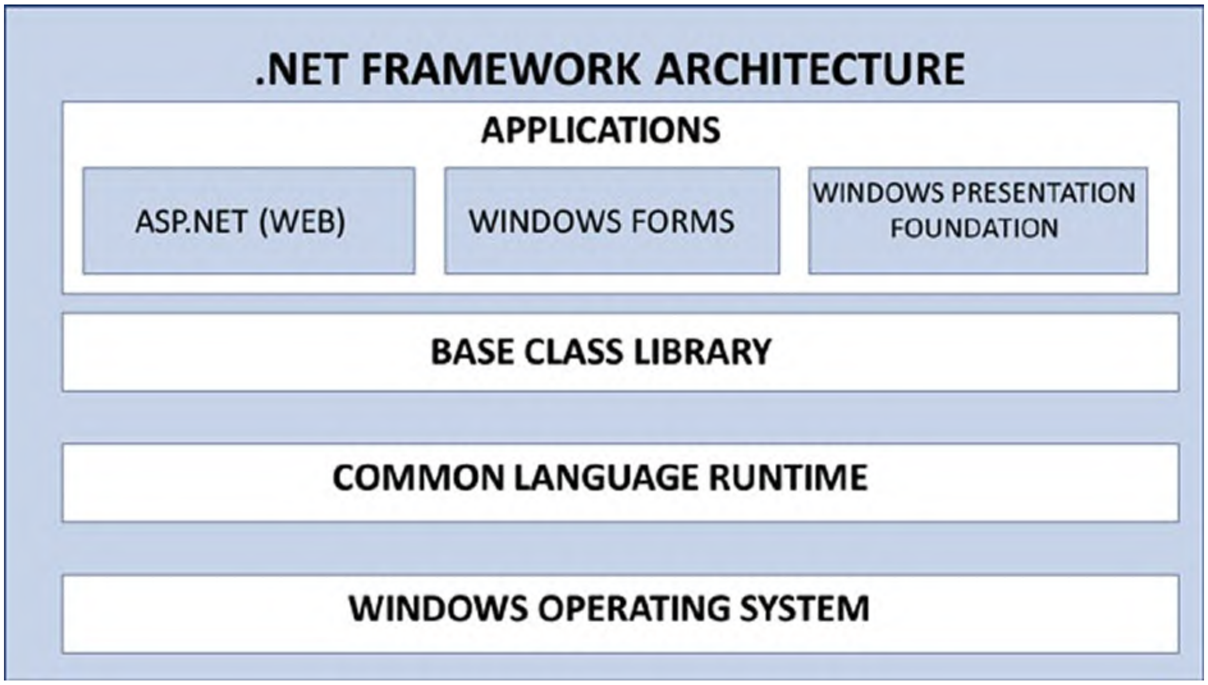


***Figure 3.1:*** *The architecture of the .NET Framework*

Upon Windows, the first layer is the **Common Language Runtime** (**CLR**). This is discussed in further details in the upcoming sections, but in short, it represents the execution environment for applications. Upon the CLR, you find the **Base Class Library** (**BCL**). This provides all the system and reusable .NET objects that you can leverage in your code. The BCL also works as the infrastructure for most .NET development platforms, such as Windows Forms, WPF, and ASP.NET, which are the last level of this layered architecture.

## The Common Language Runtime (CLR)

One of the biggest advantages of .NET is that languages like Visual Basic, C#, and F# all share the same infrastructure. This is represented by the CLR, which is responsible for:

- Managing the application execution from startup to shut down.
- Managing memory and resources used by applications.
- Managing access to system resources.
- Managing security.

The recurring word managing in the list above is not casual. In fact, when talking about .NET, you will often hear the word managed, referred to a *managed environment* and *managed cod*e. For better understanding, let’s consider how developers could build applications for Windows before .NET. A developer was fully responsible for *managing system* resources accessed by an application, memory allocation, files and so on, and the main reason for this is that applications could directly access the system. This can have dangerous implications if resources are not handled carefully. The managed environment provided by .NET makes sure that an application talks to the .NET Framework via CLR rather than talking directly with the system. The CLR also takes care of managing memory on behalf of the application, for example, deallocating objects that are no longer needed. In general, the CLR takes care of most system resources an application needs, and it can grant access to system resources only if an application is fully trusted. Obviously, there might still be situations where developers need to leverage system resources directly, so .NET makes it possible to access the Windows API via the so-called **P/Invoke**, which stands for Platform Invoke. P/Invoke allows for writing the so-called **unmanaged code**, a technique that should be only used when .NET does not provide an API for a given task. This is also the reason why P/Invoke is not discussed in this book.

## Compilers and the concept of Assembly

You create applications writing source code with a programming language, but the operating system cannot directly understand what you write. For this reason, development environments include compilers. A compiler is a piece of software that translates the source code into machine language that an operating system can understand. For example, .NET ships with several compilers: the C# compiler, the Visual Basic compiler, the F# compiler; you can write code with either C#, Visual Basic or F# programming languages, and then the related compiler translates the source code into a binary file that the system can understand and run, and this also applies to mobile systems.

**There would be much more to say about the compilation process and the additional tools, like linkers, invoked by the development environment to produce an executable file. Regarding Xamarin, you will get all the information in the upcoming chapters. About other development platforms, it is not possible to explain everything in this book, and it would also be out of scope, so the best thing to do is visit the official Microsoft documentation (https://docs.microsoft.com/) and read how it works in the appropriate platform documentation.**

Mobile applications and desktop or Web applications obviously have different architectures, so the compilation process works a bit differently. However, some general concepts in .NET are important for you to know, and they also apply to Xamarin. Generally speaking, programmers write code with a high-level programming language, and then source code is parsed and analyzed by compilers. These are responsible for generating binary files that an operating system can run. Binary files (or shortly binaries) can be executable files or dynamic link libraries (`.dll`). Binaries usually need a runtime, a set of libraries that contain all the types of the application needs. The key point is that applications written with Visual Basic 6 require their own runtime, applications written with C++ require another runtime, and applications written with Java require yet another runtime. One of the goals of .NET is to simplify how the compilation process happens over source code written with different languages so that compilers of all the .NET languages can produce binaries that run on the CLR.

## Assemblies in .NET

Regardless of the language you choose to write code, in .NET, all compilers generate an assembly. You can think of an assembly as a binary file containing CIL code and metadata. CIL is the acronym for Common

Intermediate Language, which is an evolved, object-oriented assembly programming language. CIL's instructions are CPU-independent, and the key point is that a code snippet written with any .NET language will result in the same CIL code. The other piece of an assembly is metadata. This includes information about the types and members defined in your code, digital signatures and references to other types defined externally. This is useful for the CLR to identify the needs of your application in advance. This also happened with Mono, until .NET 6 unified all platforms under the same umbrella. So, despite the **`.exe`** or **`.dll`** extension, a .NET assembly is not really an executable that can be immediately run by the operating system. When you start a .NET executable, Windows understands that it is an assembly and invokes the **just-in-time** (**JIT**) compiler. JIT is responsible for analyzing the assembly's metadata, packaging the information, and compiling on-the-fly the CIL code into machine code that the operating system can interpret.

### The Base Class Library

Another important building block of the .NET Framework is the **Base Class Library** (**BCL**). The BCL implements hundreds of thousands of reusable objects and API that you can use in your code and that are available to all the .NET languages and to most of the development platforms you use to build applications, such as (but not limited to) Windows Forms, WPF, ASP.NET, and Xamarin. Leveraging types defined in the BCL allows you to perform a wide number of programming tasks writing managed code. Types are further discussed in *Chapter 4: The C# Programming Language*, but in short, they are a way to represent an object. The BCL group types within namespaces and the name of each namespace usually refers to the technology. For example, the **`System.Windows`** namespace exposes types and additional namespaces to support the development of applications with **Windows Presentation Foundation** (**WPF**), whereas the **`System.Web`** namespace exposes types and additional namespaces to support the development of web applications with ASP.NET. You can easily recognize namespaces defined in the BCL because their name begins with the **`System`** prefix.

## From .NET Framework to .NET Core

In previous releases, the .NET Framework was a unique environment for building a variety of applications, including desktop applications, web applications, and apps for mobile devices with flavors like the .NET Compact Framework. In the last few years, Microsoft has taken some steps forward: they have kept the full .NET Framework for desktop applications, and they have created .NET Core (**https://github.com/microsoft/dotnet**). This is an open-source, general-purpose, and modular subset of the .NET Framework that is designed to be portable across platforms, with the goal of maximizing code reuse and sharing. .NET Core is modular because it is offered in smaller assembly packages with mostly no dependencies rather than one large assembly that contains most of the core functionality. This is important for two reasons: Microsoft could update .NET Core with an agile development model, while you as a developer can simply choose the functionality pieces that you need for your apps and libraries. Instead of adding assembly references, as you are used to doing if you have existing experience with Visual Studio, when developing for .NET Core, you get packages via the NuGet Package Manager, an integrated tool in Visual Studio 2019 that makes it easy to download and include libraries in your projects.

## Advantages of .NET Core

The first advantage of .NET Core is that it is cross-platform. This means that it runs not only on Windows but also on macOS and Linux and its derived distributions. This is a true revolution for Microsoft, which makes it possible for developers with experience on the Microsoft stack to write and deploy C# code on different systems. With .NET Core, you can create the following applications:

- Universal apps for Windows 10
- Cross-platform web applications with ASP.NET Core
- Portable class libraries
- Portable Console applications

In terms of architecture, .NET Core includes two major components:

- **CoreCLR**, a small portable Runtime that includes the garbage collector and just-in-time compiler (RyuJIT), but it does not include features like application domains or code access security. The runtime is delivered via NuGet and is currently represented by the **`Microsoft.CoreCLR`** package.

- Base class libraries, which offer almost the same code as the full .NET Framework BCL, but they have been refactored to remove dependencies so that it is easier to enable a smaller set of libraries.

Applications built with .NET Core run in an isolated environment. Visual Studio packages the CoreCLR, the application package, and libraries used by your application into one local package. With this approach, your applications are not affected by machine-wide versions of the full .NET Framework, and, most importantly, they have no dependencies on the operating system. And because .NET Core could run on Windows, Mac OSX, and Linux, it provides a shared implementation of APIs exposed by each operating system so that you can write the same code, regardless of the platform the application will run on. These shared implementations of the API must adhere to the .NET Standard (**https://docs.microsoft.com/en-us/dotnet/standard/net-standard**), a formal specification that establishes how an API must be implemented in order to be available on multiple platforms. *Figure 3.2* provides a representation of the .NET Core 5 architecture:

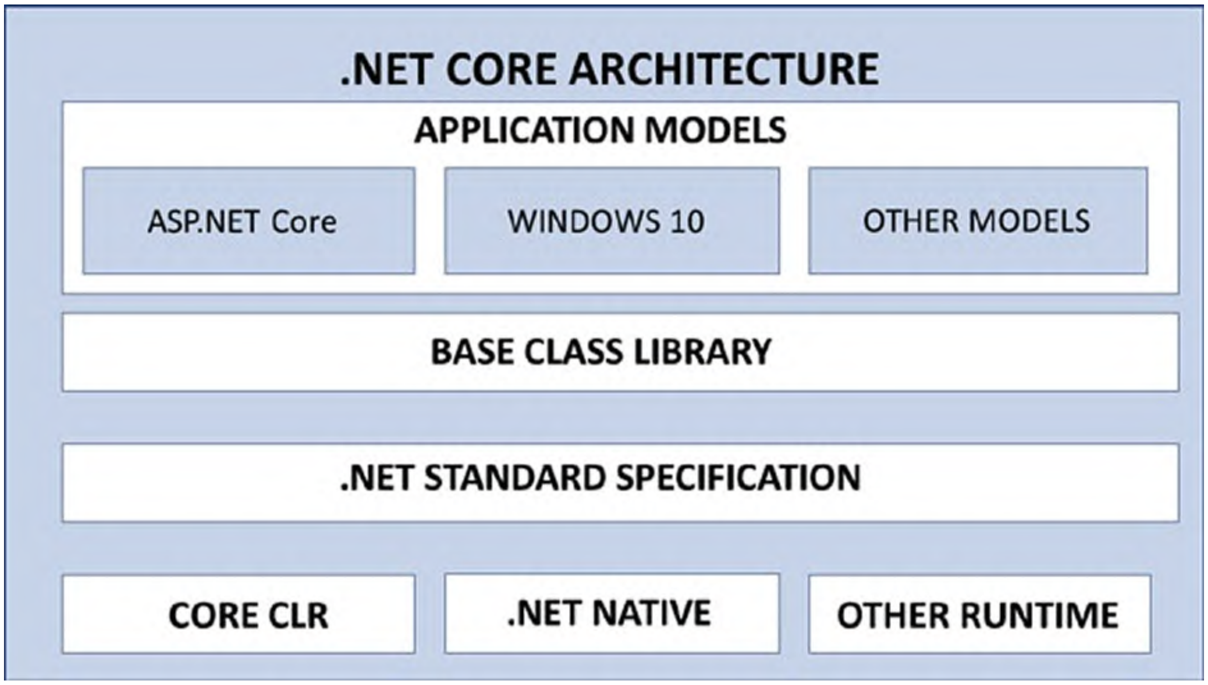


***Figure 3.2:*** *The architecture of .NET Core*

Now, the point is that there are two major platforms, .NET Framework and

.NET Core, plus additional development technologies like Xamarin that still work with .NET languages (C# and F#), with most of the .NET API but on a different runtime (Mono in this case). This is the point where .NET 6 comes in to unify them all.

## .NET 6: One .NET

.NET Framework, .NET Core, and Xamarin share several APIs, libraries, tools, and programming languages but each targets different scenarios. For Xamarin, the underlying runtime is also different, because it is based on Mono. With this in mind, Microsoft decided to unify all these platforms into one .NET technology, with one set of API, libraries, and tools. This unification goes by the name of .NET. The work of bringing all the platforms under one .NET started with .NET 5, released in October 2020, whose main goal has been to make Xamarin use .NET Core instead of Mono and have one codebase for both. For Microsoft, the next step is to unify the .NET Framework and .NET 5 into one shared set of APIs, which goes under the name of .NET 6, released in November 2021. *Figure 3.3* shows how the .NET 6 architecture appears:

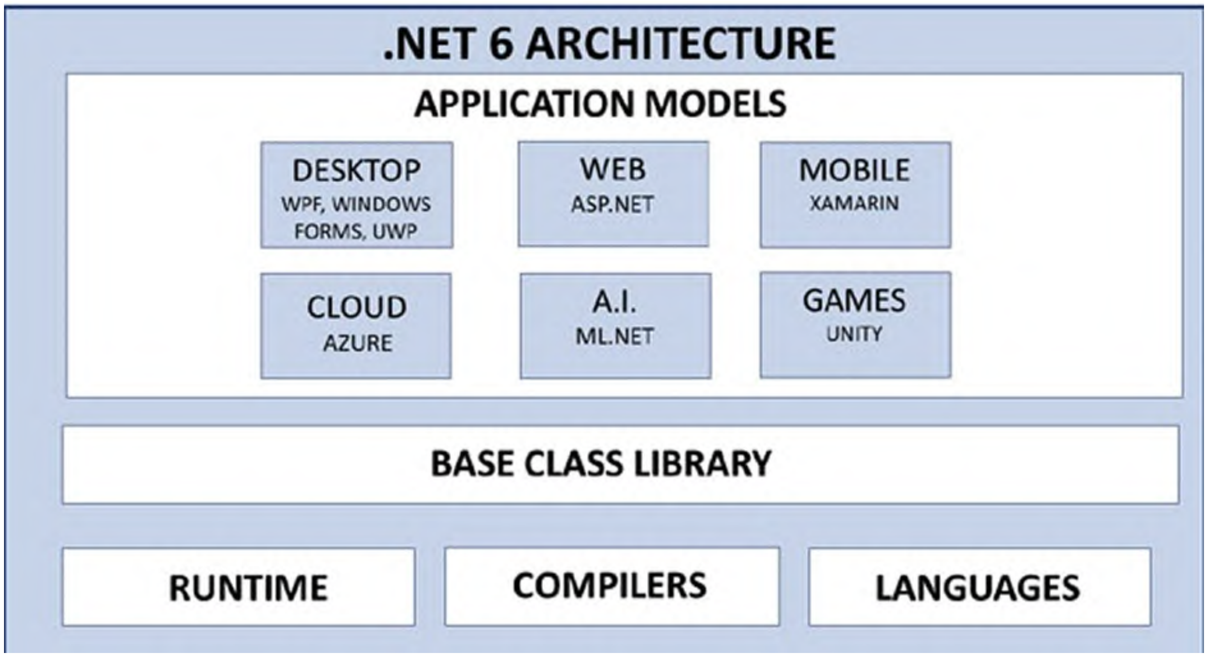


***Figure 3.3:*** *The architecture of .NET 6*

As you can see, all the high-level development platforms now rely on the same runtime, libraries, API and tools. Obviously, .NET 6 improves productivity for developers who now have to deal with one platform, and it also provides general performance improvements and perfectly integrates with Microsoft Visual Studio to offer the best development experience possible. When it comes to Visual Studio, it is now time to discover how you can build applications with this development environment and most of the features you will use when working with Xamarin and Xamarin.Forms.

## Building applications with Visual Studio

You develop applications in C# by using the Microsoft Visual Studio IDE. The latest version available is 2022, released in November 2021, but the minimum recommended version to build apps with Xamarin is 2019. Figures in this chapter are based on Visual Studio 2019, but they appear the same with Visual Studio 2022 as well. The goal of this section is to provide the necessary knowledge of the tools you need in order to work with Xamarin. It is worth mentioning that Visual Studio is a complex environment, and therefore, this chapter can only provide an overview of the most common tasks you will perform from within Visual Studio and of the most relevant tools you need. In short, you will get familiar with the IDE. The tools described in this chapter are common to all the .NET development platforms. Tools that are specific to Xamarin.Forms will be introduced in the upcoming chapters, where appropriate. At this point, you can start Visual Studio. In the Windows' Start menu, you can find a shortcut called **Visual Studio 2019** (or 2022 if this is the version you installed). Click on it to start the development environment.

## Signing in with a Microsoft account

The first time you start Visual Studio, you will be asked to sign in with a Microsoft Account. This is usually an email address based on one of the Microsoft providers, such as Hotmail, Live, Outlook, but it can also be an address from another provider and associated to Outlook as a Microsoft Account. *Figure 3.4* demonstrates this:

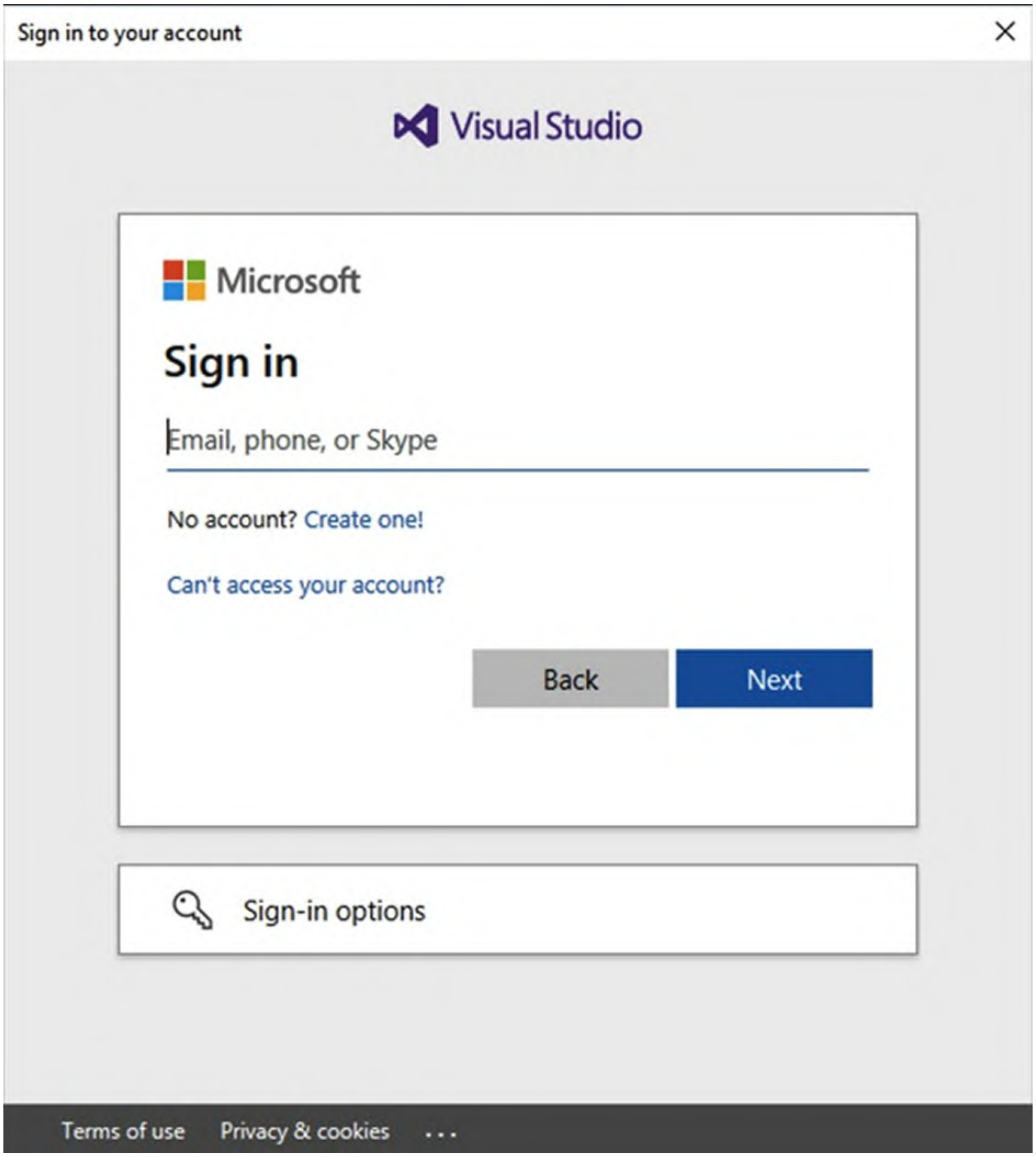


***Figure 3.4:*** *Signing into Visual Studio*

Though not mandatory, it is strongly recommended that you sign in to Visual Studio for the following reasons:

- Settings synchronization across machines is enabled.
- Visual Studio Community is permanently unlocked.
- The IDE will automatically log in to all the Microsoft services

connected to the specified Microsoft Account.

- Visual Studio is fully unlocked if it was downloaded from an MSDN subscription.

Click on **`Sign in`** and enter your email address; then, click on **`Next`**. At this point, enter your password when and click on **`Sign In`**. Visual Studio will now complete the login process to the associated Microsoft services, and it will show the Start window.

## Synchronized settings

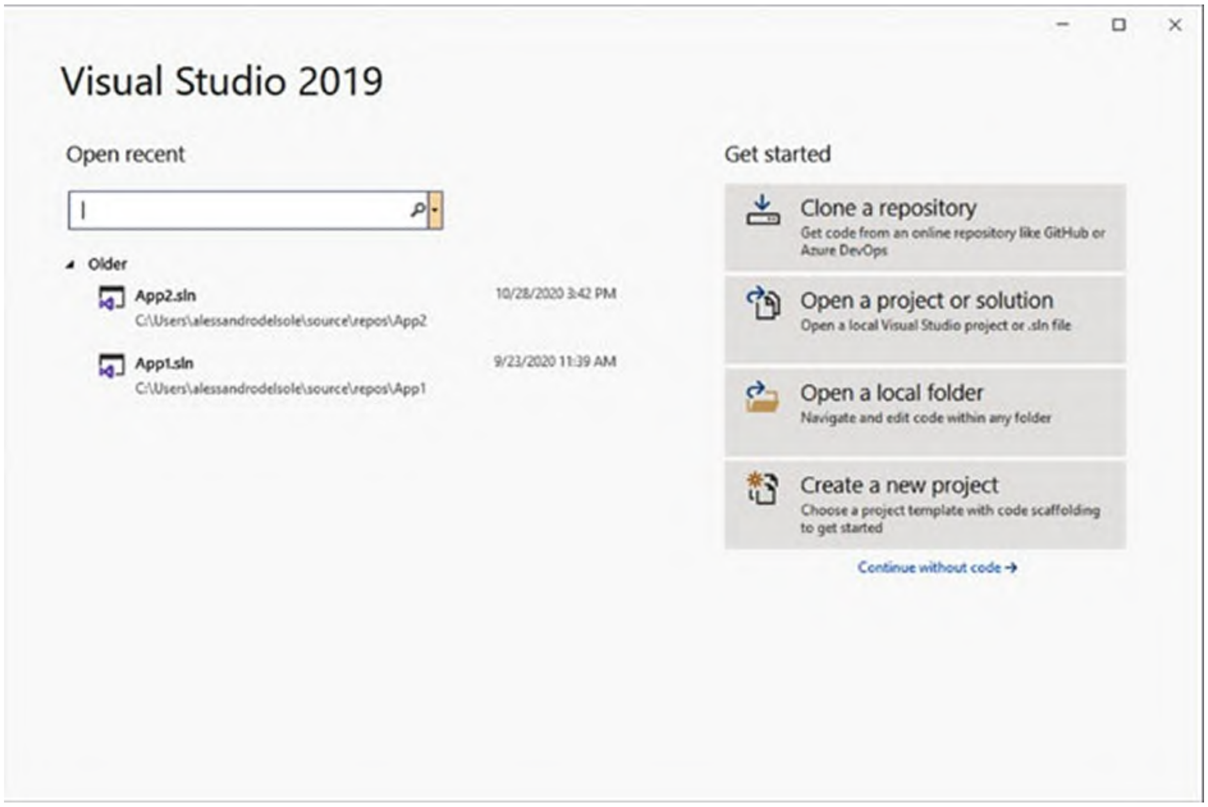
One of the biggest advantages of connecting a Microsoft Account to Visual Studio is enabling a feature known as **Synchronized Settings**. As you discover in the book, the IDE is highly customizable, and you might want the same preferences on all the machines where you have installed Visual Studio. More specifically, synchronized settings are part of Visual Studio's general options, which you access by selecting the **`Tools`** menu and then the **`Options`** command. These include the following:

- **Theme settings**: You can change the graphic theme of Visual Studio by opening the Environment page, and finally, the **`General`** tab of the **`Options`** dialog.
- **Startup settings**: These can be customized by opening the **`Startup`** tab of the **`Options`** dialog.
- All the available settings for the text editor. These can be retrieved and customized by opening the **`Text Editor`** tab in the **`Options`** dialog.
- All the available settings for fonts and colors available in **`Environment`**, **`Fonts`**, and **`Colors`** tab of the **`Options`** dialog.
- All the available keyboard shortcuts, both default and custom, available in the **`Environment`**, **`Keyboard`** tab of the **`Options`** dialog.
- Custom command aliases.

Some of these settings are also described in the upcoming sections.

# Introducing the Start window

When Visual Studio 2019 starts up, it shows the Start window. *Figure 3.5* shows an example based on Visual Studio 2019:


Visual Studio 2019

Open recent

Older

App2.sln 10/28/2020 3:42 PM
C:\Users\alessandrodelsole\source\repos\App2

App1.sln 9/23/2020 11:39 AM
C:\Users\alessandrodelsole\source\repos\App1

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

***Figure 3.5:*** *Visual Studio 2019's Start page*

On the left side, the Start window shows a list of recently opened projects. You can remove one or more items from the list by right-clicking on them and selecting **`Remove From List`**. On the right side, the Start window provides shortcuts to common actions, such as the following:

- **Clone or check out code**: This shortcut allows you to connect to a Git repository on Azure DevOps or GitHub so that you can clone the repository or check out the code. This feature is discussed in further detail in the next section.
- **Open a project or solution**: This shortcut allows you to open an existing solution or project.
- **Open a local folder**: This shortcut makes it possible for opening an existing folder as a loose assortment of files, similar to what Visual Studio Code also allows.
- **Create a new project**: This launches the same-named dialog, where you can create a new project by selecting a template. This feature will be discussed thoroughly in the next section.

If you do not want to work with projects and simply want to open the IDE, you can click on the **`Continue without code hyperlink`**. You will create a new project soon, but it is important for you to first know how projects are structured in Visual Studio and .NET.

## Understanding projects and solutions

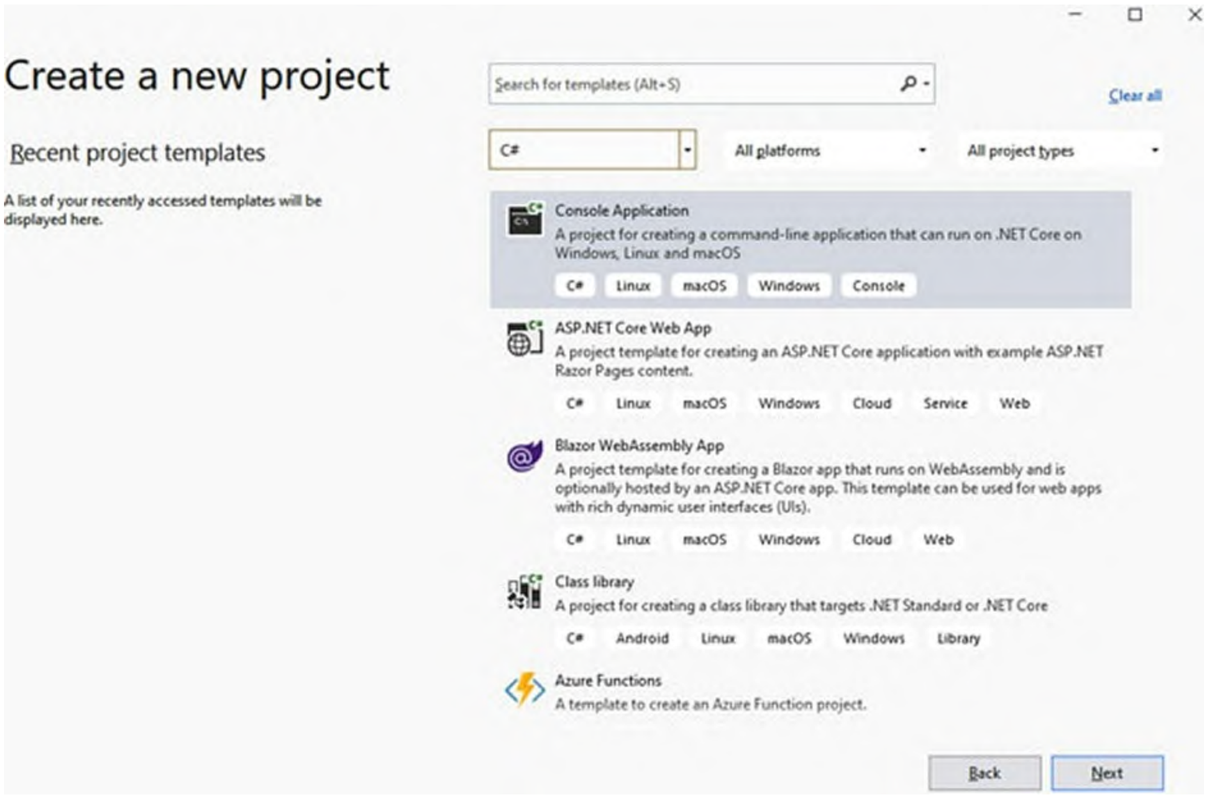
In modern software development, an application is not simply the result of compiling just one code file into a binary. Instead, you write many code files that are linked to one another, and you need images, data files, fonts, and other resources. All the files required to build an application are collected in a project. In Visual Studio, you can create dozens of different project types, depending on the kind application you want to build. In C#, the list of files and resources required for a project is contained inside a **`.csproj`** file. An **Extensible Markup Language** (**XML**) file that Visual Studio knows how to handle to properly load a project. In Visual Studio, you might need to work with multiple projects together, which is a very common situation. This is represented by solutions that can be considered a container for projects. One solution can contain an infinite number of projects of different types, such as C# projects, class libraries, Web services, and Windows client applications. A solution is an **`.sln`** file that, similar to **`.csproj`** project files, has an XML structure. Visual Studio knows how to parse the **`.sln`** file and loads projects, accordingly, providing the option to manage projects and solutions via the Solution Explorer tool window, which will be discussed in the upcoming sections.

**Due to the large number of different project that you can create with Visual Studio, in this book it is not possible to describe all the possible options. For this reason, the focus will be on projects you can create with C#, since this is the language, you also use to work with Xamarin.**

### Creating projects with C#

You create projects by clicking on the **`Create a new project`** shortcut in the Start window or via the **`File`**, **`New Project`** command when already in the IDE. This will then show a full list of supported project templates for all languages and platforms. The list can be filtered by using one or more of the dropdowns at the upper-right corner of the dialog. For example, click on the

**`All languages`** dropdown and select **`C#`** to restrict the list based on this language (see *Figure 3.6*). The list of available project templates may vary depending on which workloads you have selected during the installation of Visual Studio, but in general, you can create projects to create desktop applications, web applications and services, mobile applications, and reusable libraries.


Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

Clear all

C# | All platforms | All project types

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.

***Figure 3.6:*** *Creating projects with C#*

*Table 3.1* summarizes a list of the project templates for C# that are most relevant for you as a Xamarin developer. You can see each template by scrolling through the list in Visual Studio:

| **Template** | **Short description** |
|---|---|
| Console Application | A project for creating a command-line application that runs on different systems. |
| ASP.NET Core Web App | A project for creating Web applications that can be hosted on different systems. |
| Class library | A project for creating a reusable library of code. |
| ASP.NET Core Web API | A project for creating a Web service that exposes API in a Representational State Transfer (REST) approach. |

| | |
|---|---|
| Windows Forms App | A project for creating a desktop application upon the Windows Forms technology. |
| WPF Application | A project for creating a desktop application upon the Windows Presentation Foundation technology. |
| Blank App (Universal Windows) | A project for creating applications that target the Universal Windows Platform. |
| Mobile App (Xamarin.Forms) | A project template with multiple projects that allows for creating apps for iOS, Android, and UWP using Xamarin.Forms. This is the project template you will use most in the book. |
| Android App (Xamarin) | A project for creating Android apps with Xamarin.Android. |
| iOS App (Xamarin) | A project for creating iOS apps with Xamarin.iOS. |
| Android Wear (Xamarin) | A project for creating Android apps for wearable devices with Xamarin.Android. |
| iOS Wear (Xamarin) | A project for creating iOS apps for wearable devices with Xamarin.iOS. |
| Xamarin.UITest Cross Platform Test Project | A project that allows implementing automated tests for the user interface over Xamarin projects. |

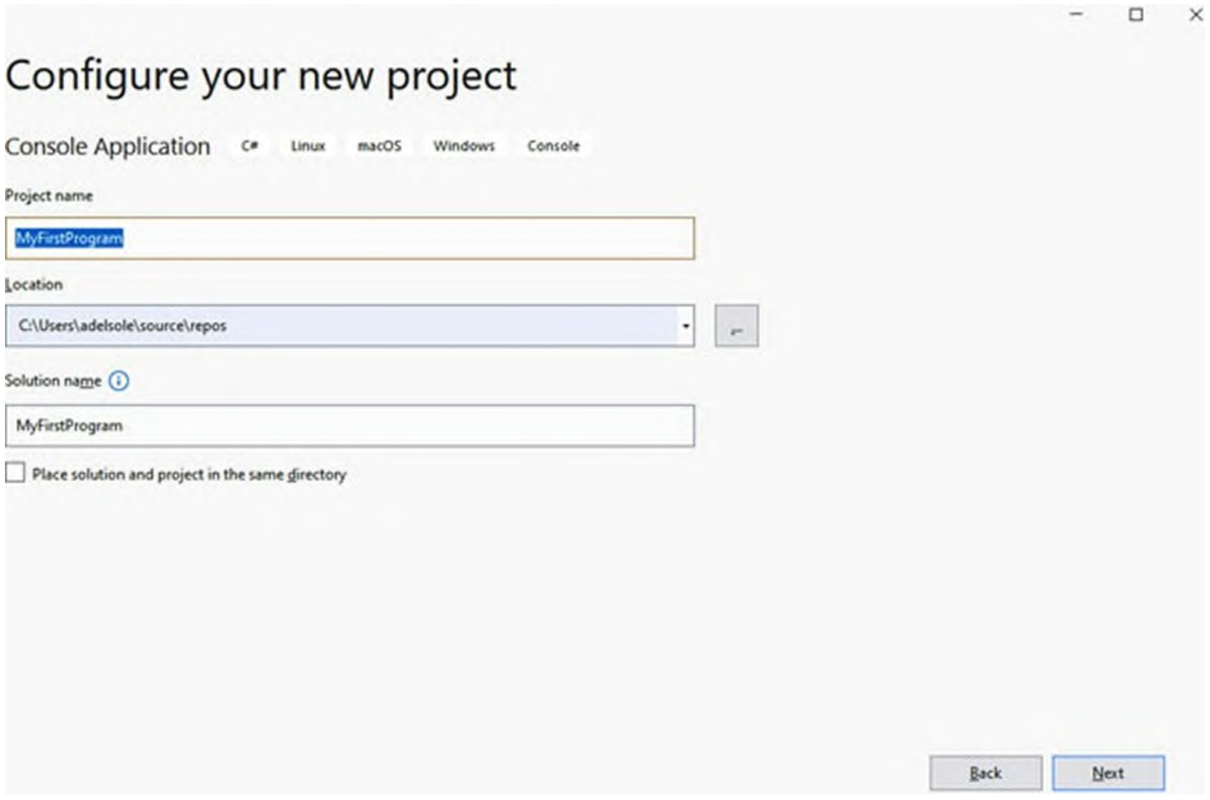
***Table 3.1:*** *Most relevant project templates for C#*

**A short description is available for every project template in the Start window, which is visible below the project name (see *Figure 3.6*).**

In this chapter, you will use the Console application template to discover tools in Visual Studio. In the rest of the book, you will use the Mobile App (`Xamarin.Forms`) project template to create mobile apps. When you create a new project, Visual Studio first generates a solution containing one or more projects, depending on the selected template. Many concepts will be clarified in the next section, where you create your first project with Visual Studio.

## Creating your first C# project

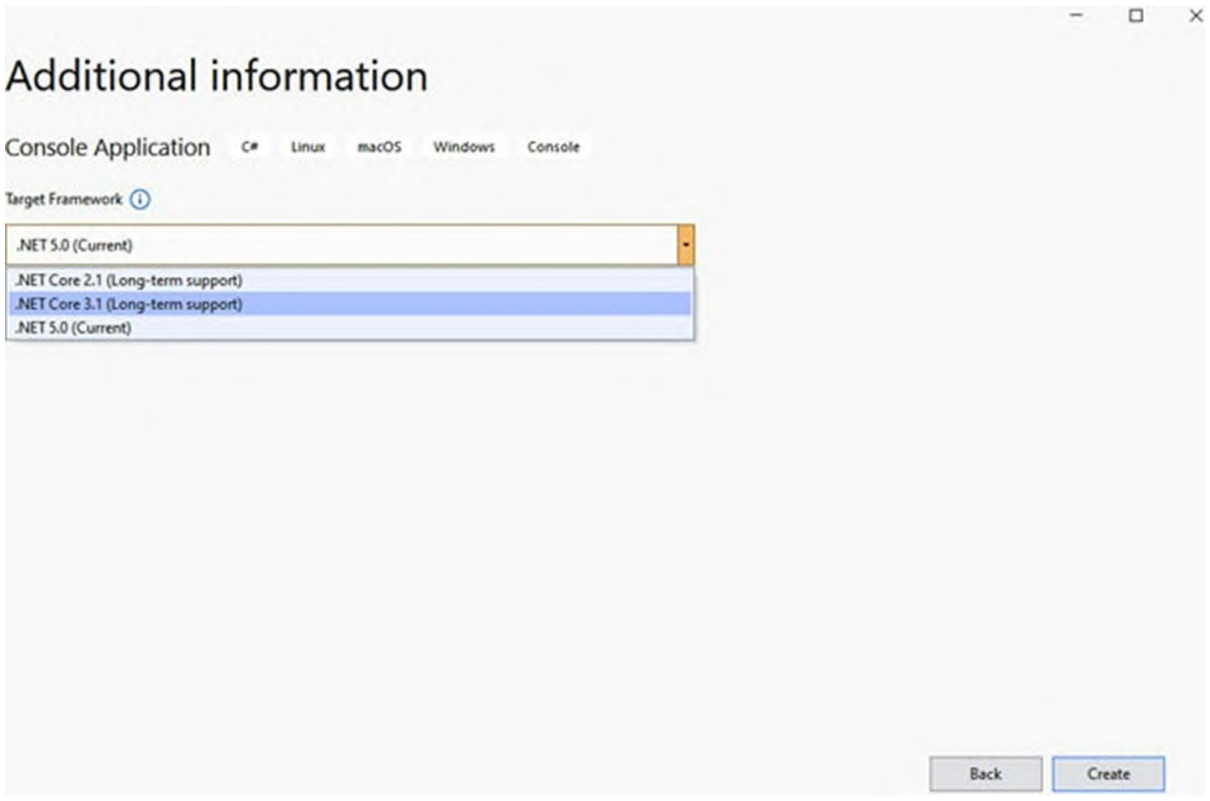
Now that you have seen the main concepts about creating projects, you are ready to create your first C# project. You will work with a Console application, which is optimal for instructional purposes and that allows for quickly understanding what you do without relying on a specific platform. In the **`Start`** dialog, click on the **`Console Application`** template. Visual Studio will show the **`Configure your new project`** dialog, where you will be able to specify the project name and location on disk (see *Figure 3.7*). The project name cannot contain blank spaces. Enter **`MyFirstProgram`** as the name in the Project name box. Visual Studio provides a default location to store projects, which is usually under **`C:\Users\UserName\source\repos`**, where **`UserName`** is your Windows user. You can provide a different location, but for now. it is okay to use the default one. When ready, click on **`Next`**:



Project name
MyFirstProgram
Location
C:\Users\adelsole\source\repos
Solution name
MyFirstProgram
Place solution and project in the same directory

***Figure 3.7:*** *Supplying project information*

In the **`Additional information`** dialog (see *Figure 3.8*), you will be able to specify the target framework. Select the highest .NET Core version available. On my machine, it is .NET 5.0 (**`Current`**). When done, click on **`Create`**:



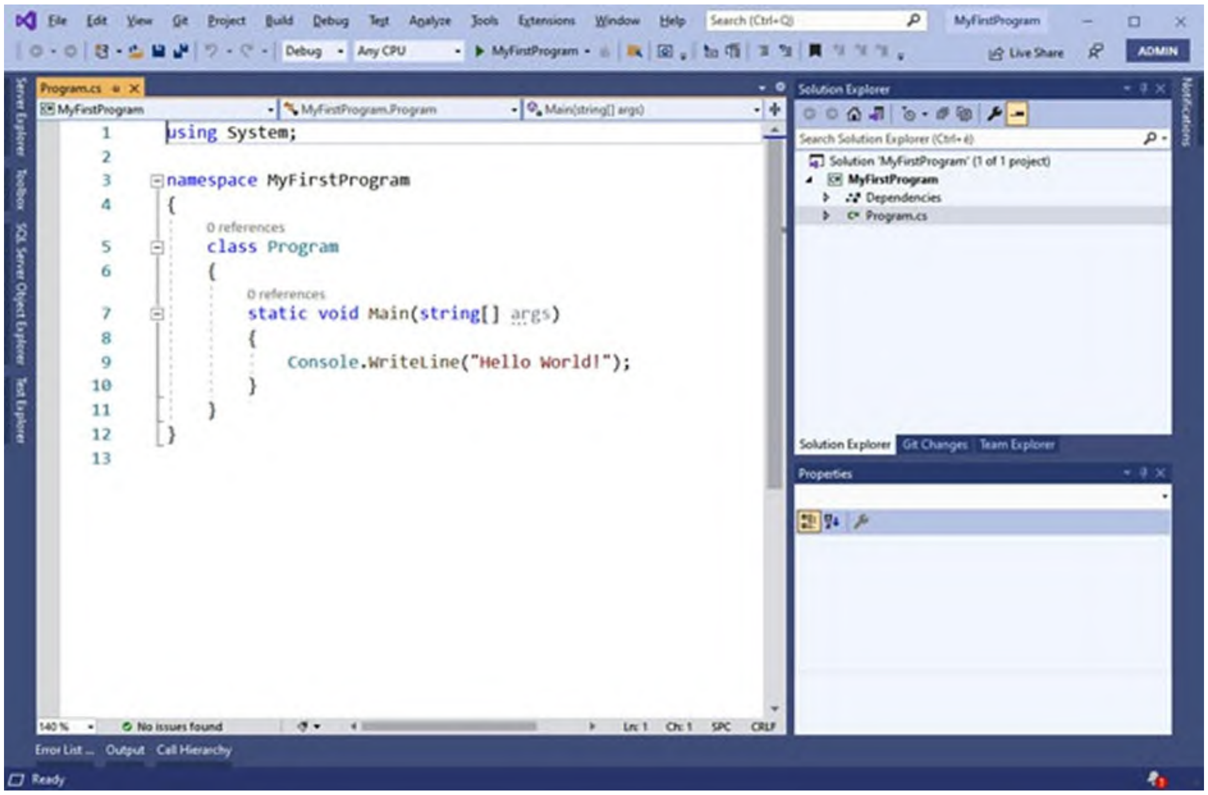
Target Framework

.NET 5.0 (Current)

.NET Core 2.1 (Long-term support)
.NET Core 3.1 (Long-term support)
.NET 5.0 (Current)

***Figure 3.8:*** *Supplying project information*

After a few seconds, your project is ready, and Visual Studio 2019 shows the code editor (see *Figure 3.9*):



```
using System;

namespace MyFirstProgram
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

***Figure 3.9:*** *The new project is ready*

The code simply shows a message in the system Console window, and this simple project will serve as the base for learning tools and features in Visual Studio 2019 that you need to know as a developer.

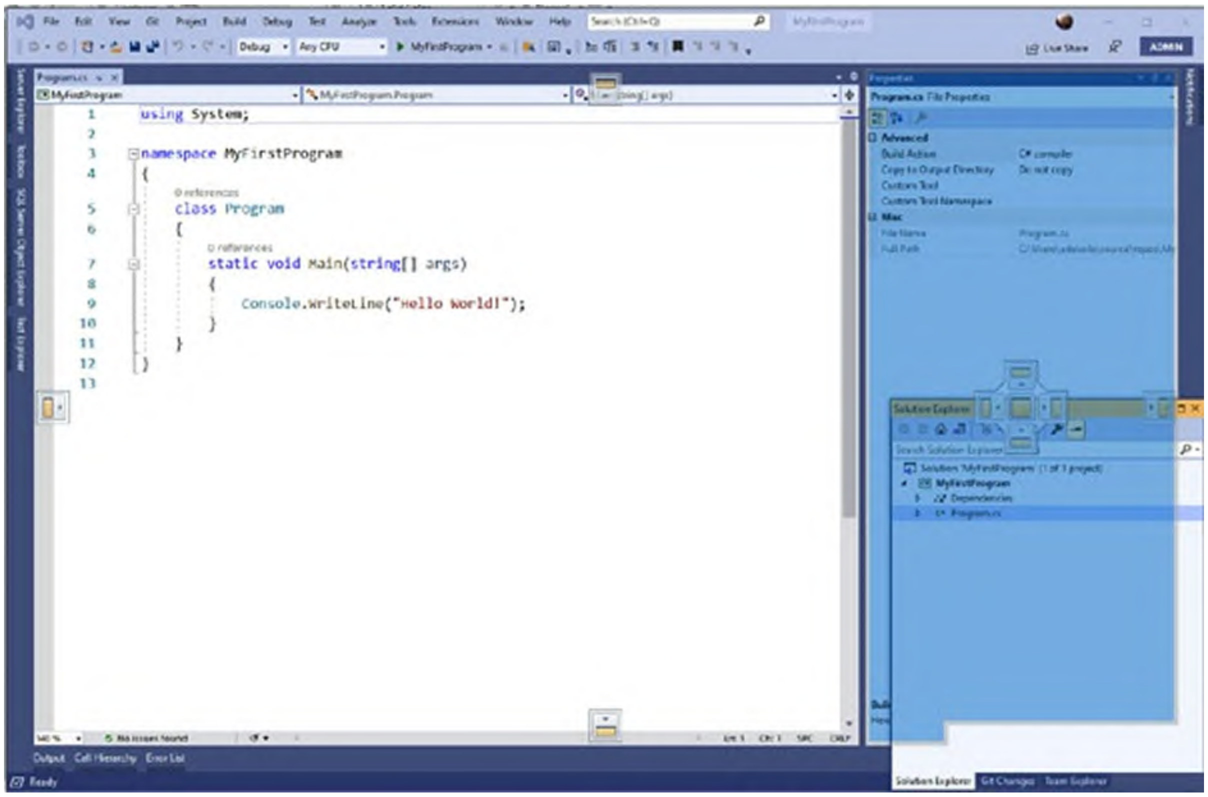
## Working with the Visual Studio IDE

The Visual Studio 2019 IDE is a very powerful development environment, which offers a large number of productivity tools that will simplify your developer life. This section describes the most important tools, commands, and features that you need to know to successfully understand the topics covered in the upcoming chapters. Some other tools require more knowledge of the C# programming language and will, therefore, be discussed in the later chapters, where appropriate.

## Working with tool windows

Tool windows are floating windows that can be moved, arranged, and docked to the IDE interface, which provide a large variety of tools. As a general rule, all the tool windows available in Visual Studio are listed in the **`View`** menu.

There are exceptions; for example, tool windows providing testing tools can be found in the **`Test`** menu and those that provide debugging tools can be found in the **`Debug`** menu. The examples in this book require using several tool windows, and this chapter provides an overview of the ones that are most frequently used. In particular, this chapter describes the **`Solution Explorer`**, **`Error List`**, **`Properties`**, and **`Output`** windows because you will use them very often in all your projects. You'll learn about other tool windows that are relevant to mobile app development in the appropriate chapters. Rearranging tool windows is accomplished by clicking on their title and dragging and dropping it at the new desired position and onto the most appropriate arrow in the cross shown in *Figure 3.10*:



```
using System;

namespace MyFirstProgram
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

***Figure 3.10:*** *Arranging tool windows*

When you first install Visual Studio, tool windows have a default position in the IDE, but you can rearrange them at your convenience. The following sections discuss the tool windows you'll use most frequently.

## Solution Explorer

Solution Explorer is probably the tool window you will interact with the most. It allows you to manage solutions, projects, and files. It provides a structured and complete view of the solution, its projects, and files in each project. It also provides a way to add and remove files and organize files into subfolders. *Figure 3.11* shows how the newly created C# project appears in Solution Explorer.

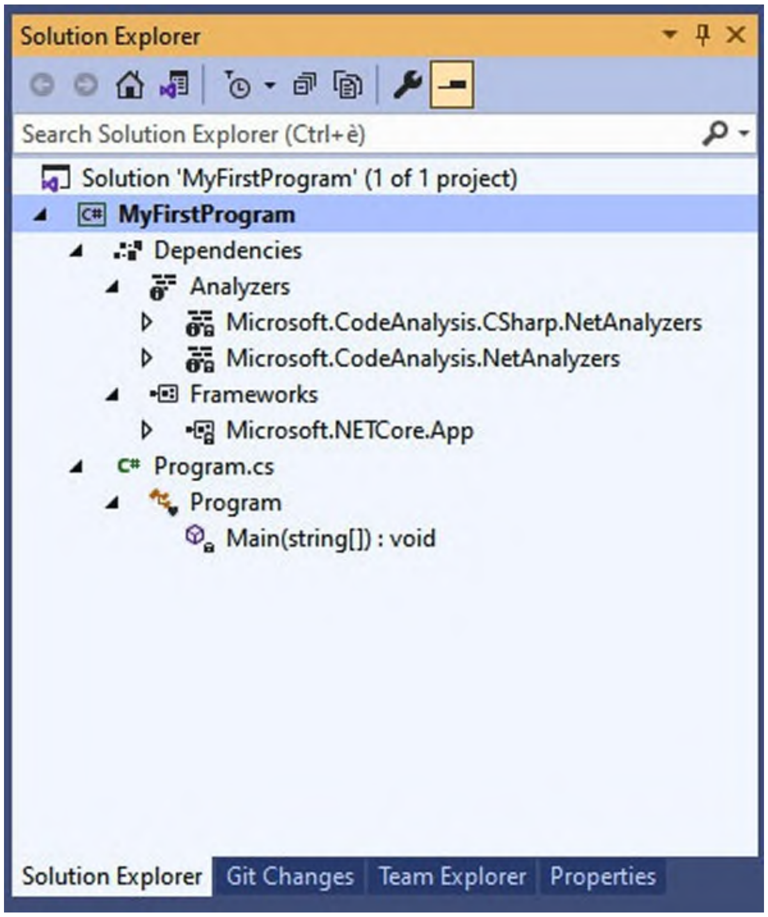


***Figure 3.11:*** *Browsing a solution with Solution Explorer*

As you can see, the solution is at the root level. Nested under it are projects (in this case, only one project). Inside each project, you can find code files and subfolders. These can contain images, database files, documents, and any other required assets. You can also get a list of all the dependencies in the project. A dependency is a library or component that is necessary for the application to work. In the case of a Console project built on top of .NET Core, the required dependency is the **`Microsoft.NETCore.App`** library that you can find under the Frameworks subfolder. In the real world, there will be several dependencies, and this will be clearer when you start working with Xamarin in practice. Analyzers are special tools that analyze your source code as you type, highlighting issues depending on rules and patterns defined by Microsoft. However, developers can also create their own analyzers to implement custom code style rules. Analyzers will not be covered in this book, but an official documentation page is available for them (**https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview**). You use Solution Explorer to add and manage items in projects as well as to see which files constitute a project. By default, Solution Explorer shows only the items that compose the project. If you need a complete view of references and auto-generated code files, you can click on the **`Show All Files`** button, which is the third from right to left in the toolbar of the window. Solution Explorer also displays the list of types and their members defined inside each code file. Simply expand the name of the code file to accomplish this. For example, in *Figure 3.11*, you can see that the **`Program.cs`** code file defines the **`Program`** class, which exposes a method called **`Main`**. When you double-click on a member, the code editor will automatically open the file and move the cursor to the member definition.

**If terms like class or member are completely new for you, do not worry. In the next chapter, you will learn everything that is required to write C# code from a beginner point of view.**

You can also filter the list of items shown in Solution Explorer by typing keywords in the search box. Adding, editing, or removing items in a project is accomplished by right-clicking on the project name and then by selecting the appropriate command from the **`Context`** menu. *Figure 3.12* shows how this appears:

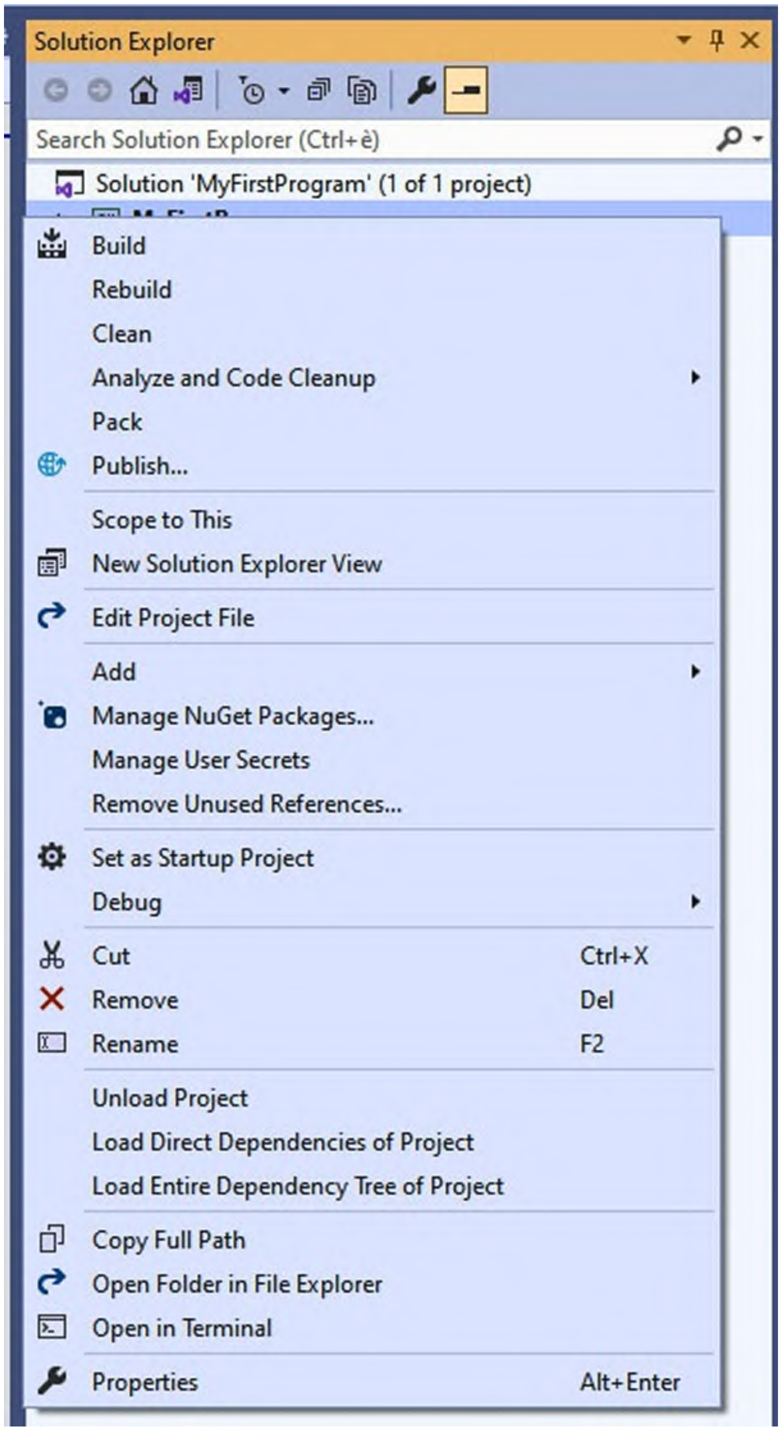
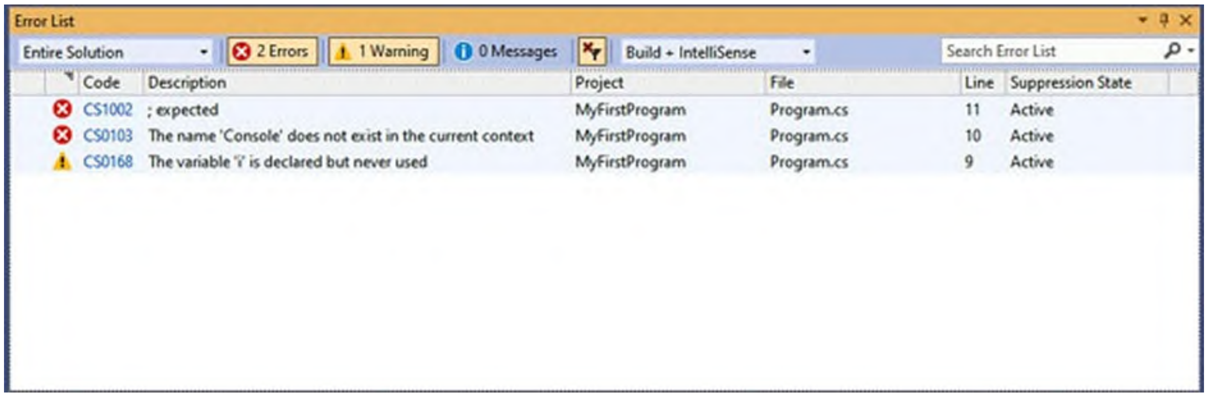


***Figure 3.12:*** *The Solution Explorer contextual menu*

As you can see in the preceding figure, the context menu offers shortcuts to many tasks that you can perform against projects or solutions. For example, if you select the **`Add`** command, you will be able to add new items via a dedicated dialog. When you will be asked to add new items to your projects in the next chapter, you can select **`Add`** | **`New Item`**. You can also execute tasks over individual files. Just right-click on items in the solution instead of right-clicking on the project's name.

## The Error List

The Error List tool window displays the list of errors, warning messages, and informational messages that Visual Studio reports during the development and compilation process of your applications. *Figure 3.13* shows how the Error List window looks:



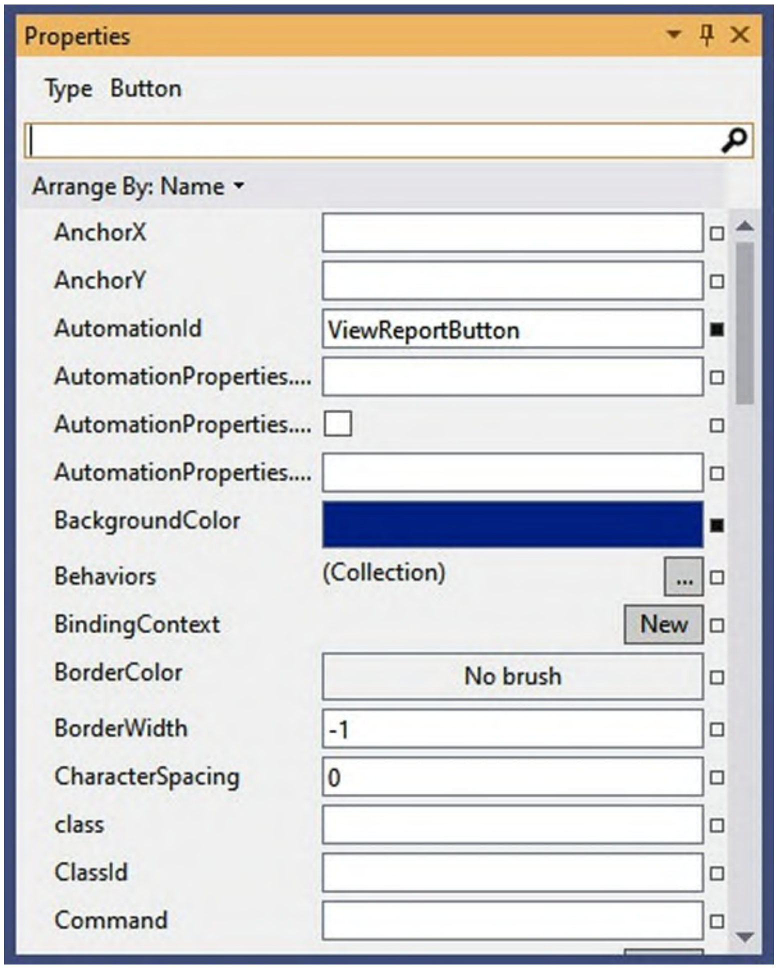
| | Code | Description | Project | File | Line | Suppression State |
|---|---|---|---|---|---|---|
| ⊗ | CS1002 | ; expected | MyFirstProgram | Program.cs | 11 | Active |
| ⊗ | CS0103 | The name 'Console' does not exist in the current context | MyFirstProgram | Program.cs | 10 | Active |
| ⚠ | CS0168 | The variable 'i' is declared but never used | MyFirstProgram | Program.cs | 9 | Active |

***Figure 3.13:*** *The Error List tool window*

In general, the **`Error List`** groups code issues that are detected in the code editor, where issues are represented with squiggles (wavy lines). Errors are highlighted with red squiggles, whereas warning messages are highlighted with green squiggles. Errors prevent your application from running, and this is the typical case of code that cannot compile successfully. They include problems that Visual Studio or the compiler encounter during the development and build process, such as syntax errors or invalid object management. Warnings are related to code that will still successfully compile but that needs your attention and that you should never ignore. For both errors and warnings, you can double-click on a message in the **`Error List`**, and the code editor will move directly to the code that caused the message. You can also press *F1* or select the **`Show Error Help`** command to get help on about a given message. Finally, informational messages just provide information and can be usually ignored.

## The Properties window

All the items in a solution have properties. These represent the characteristics of an item. You will more often hear about properties when talking about .NET objects, and the next chapter gives you an in-depth explanation, but files have properties as well. The filename is an example of property of a file. Whether you need to set object or file properties, Visual Studio offers the **`Properties`** window. *Figure 3.14* shows an example of the **`Properties`** window for a button control in Xamarin.Forms:



Type Button

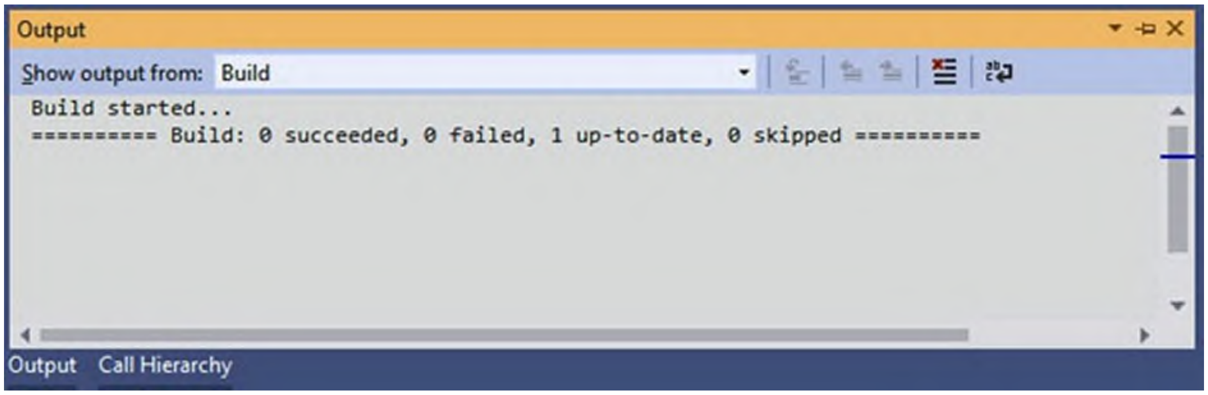
| | |
|---|---|
| AnchorX | |
| AnchorY | |
| AutomationId | ViewReportButton |
| AutomationProperties.... | |
| AutomationProperties.... | ☐ |
| AutomationProperties.... | |
| BackgroundColor | 🟦 |
| Behaviors | (Collection) ... |
| BindingContext | New |
| BorderColor | No brush |
| BorderWidth | -1 |
| CharacterSpacing | 0 |
| class | |
| ClassId | |
| Command | |

***Figure 3.14:*** *The Properties window over a Button in Xamarin.Forms*

The **`Properties`** window is made of two columns. The left column shows property names, and the right column contains the property values. For C# objects, you will normally set properties in code, but for controls (elements of the user interface) or files, you can take advantage of the **`Properties`** window. The keyboard shortcut to open the **`Properties`** window is *F4*.

## The Output window

Visual Studio continuously invokes external tools. Compilers and the NuGet package manager are examples of tools that Visual Studio often requires to run. External tools send a result of their actions when completed, and Visual Studio redirects such output to the **Output** window. Continuing the example of compilers, when the compilation process is completed, the output of the compiler is sent to the **Output** window rather than being displayed in the system console. *Figure 3.15* shows an example of the **Output** window that contains the results of the compilation process for a C# project:



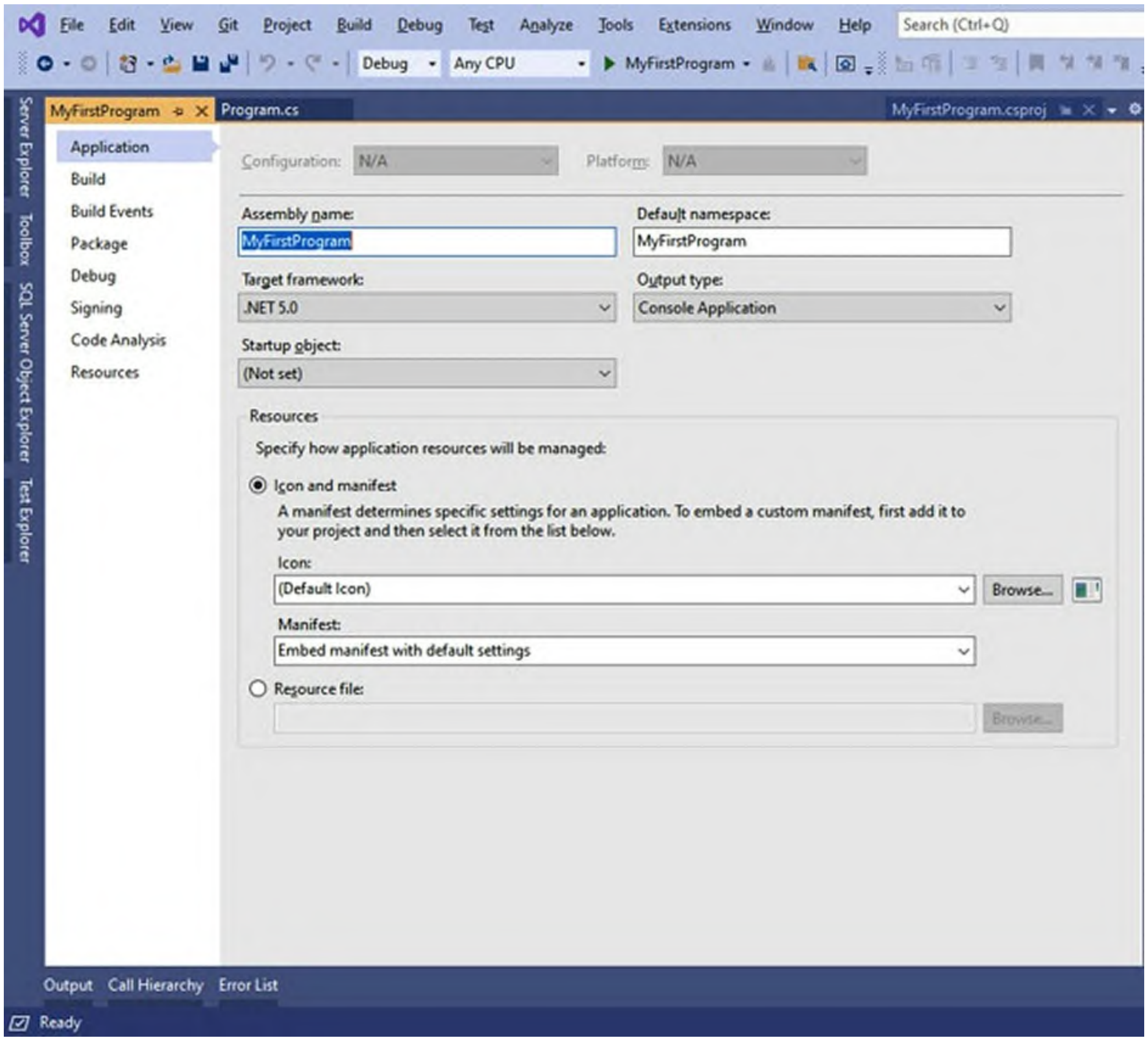
```
Build started...
========== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped ==========
```

***Figure 3.15:*** *The Output window showing the result of compiling a program*

The **Output** window is interactive. For example, when you're compiling a program, if the compiler reports any errors, the **Output** window shows them, and you can click on the buttons on the window's toolbar to navigate error messages. Navigating to error messages also makes the code editor open up the code that caused the error. You will find the **Output** window extremely useful, especially when debugging, because it also shows the debugger output. If you click on the **Show Output From** combo box, you will see a list of available outputs.

## Editing project properties

In the Visual Studio terminology, projects settings are referred to as project properties. These include the application filename, the application icon, version number, copyright information, and compilation settings. In order to open project properties, you right-click on the project name in **Solution Explorer** and then you select **Properties**. *Figure 3.16* shows the **Properties** editor for the first C# program you created previously:



Configuration: N/A    Platform: N/A

Assembly name: MyFirstProgram
Default namespace: MyFirstProgram
Target framework: .NET 5.0
Output type: Console Application
Startup object: (Not set)

Resources

Specify how application resources will be managed:

● Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

Manifest: Embed manifest with default settings

○ Resource file:

***Figure 3.16:*** *Editing project properties*

The **`Properties`** editor is organized into tabs, and each tab represents a specific area of the project, such as application-level properties, external references, deployment options, compile options, and debugging options. For now, you don't need to learn about all the tabs because they change depending on the type of project you are working on. For Xamarin projects, the **`Properties`** editor has specific tabs that will be discussed in the appropriate chapters. Actually, the **`Application`** tab is the only one that will be discussed now because it is common to all the project types.

## Editing application settings

Application settings include the executable's name, icon, or metadata that will be of interest for the operating system, such as the program version, copyright information, and so on. The **`Application`** tab allows you to edit these kinds of settings, and it is shown by default when you first open **`Properties`** (see *Figure 3.16*). More specifically:

- The **`Assembly Name`** field is used to set the name of the compiled assembly. By default, the assembly's name is based on the project name.
- The **`Default Namespace`** field is used to set the root-level namespace identifier. (Namespaces will be discussed in the next chapter). You can think of the root namespace as the object that stores everything that your project implements. According to Microsoft specifications, the root namespace should be formed as follows: **`CompanyName.ProductName.Version`** (not mandatory).
- **`Output type`** specifies the application type (for example, Console application, class library, Windows Forms application) and is automatically set by Visual Studio. You should not change the default setting in this box.
- **`Target framework`** allows you to change the target version of .NET as the runtime for the application. This might be useful with desktop applications when a new release of .NET is published, and you want to make your programs leverage its new features.
- **`Startup object`** allows you to specify the main entry point of your application. This might make sense on Windows projects, but it's not the same on Xamarin projects.

The **`Resources`** group allows you to specify an icon for Windows applications and resources that must be embedded in the application itself. These will not be used with **`Xamarin.Forms`** projects, which handle resources differently.

## Basic code editing features

The code editor is the place where you spend most of your developer life, so it is important for you to know how to leverage all the tools it offers to boost productivity. Obviously, not all the tools can be discussed in this chapter because they would require more knowledge of the C# language on one side, and more complex projects on the other side. So, you will learn about features like syntax colorization, IntelliSense, and zooming the code. Other features will be discussed in the next chapter, when you learn more about C#.

## Syntax colorization

If you look back at *Figure 3.9* and look at the source code of the sample program, you can notice how words have different colors. Visual Studio makes it visually easier to understand what each word refers to. In particular:

- Reserved words of the C# programming language are in blue.
- Names of objects defined inside .NET or custom libraries are in light blue.
- Names of members exposed by such objects are in dark grey.
- Namespace identifiers are in black.
- Strings are brown.
- Comments are in green.

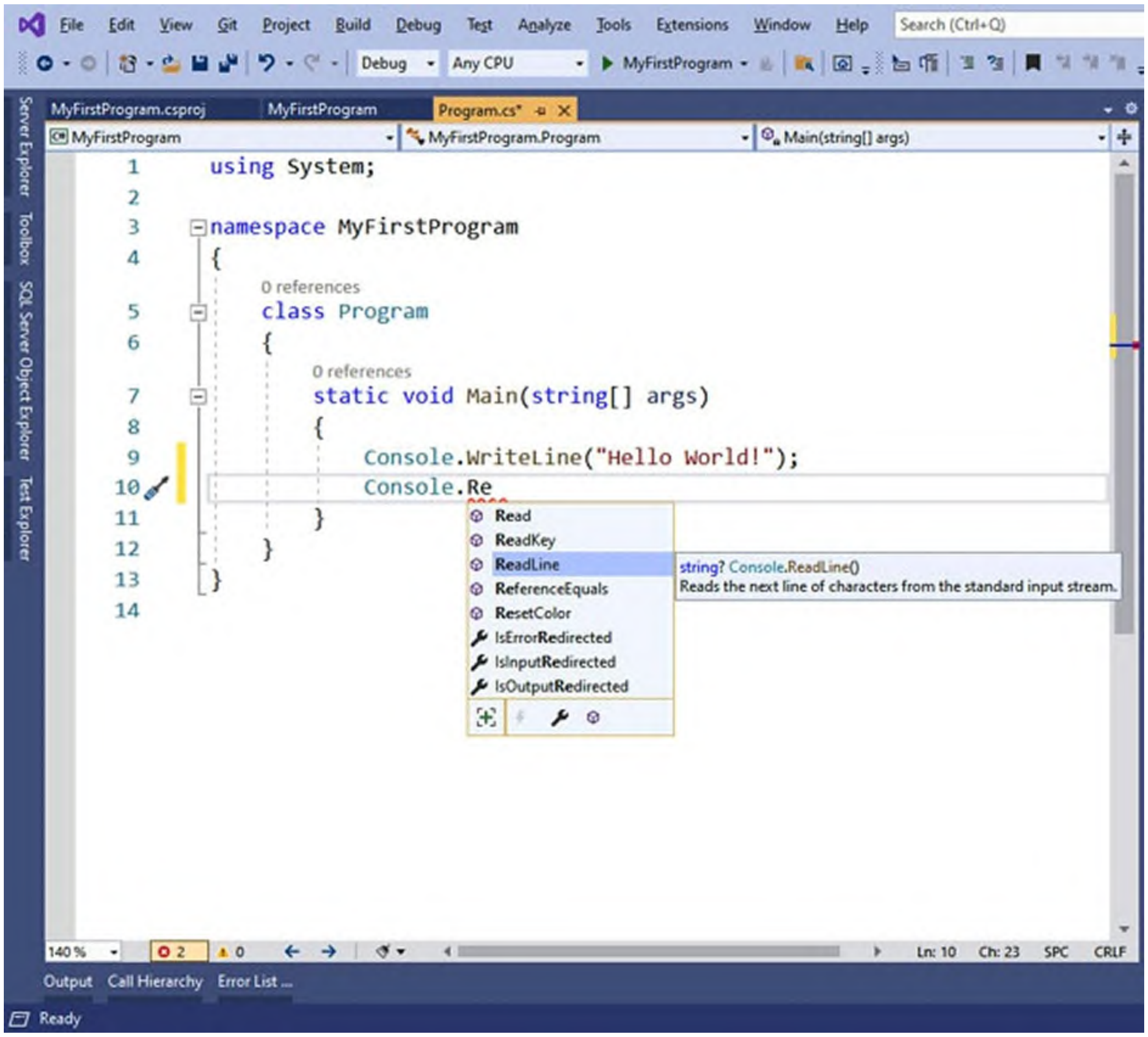
Syntax colorization is an extremely important feature for each code editor because your eye will immediately recognize parts of source code from their color, and this will make you much faster.

## Zooming the code editor

The code editor supports zooming in and out. This is accomplished by pressing the Ctrl key plus moving the mouse wheel up and down. You can also zoom the code editor by changing the zoom percentage that you can find at the bottom-left corner of the IDE (see *Figure 3.9*).

## Introducing IntelliSense

IntelliSense is a productivity tool for your coding experience represented by an integrated window that pops up in the code editor as you type, and its goal is to provide sophisticated auto-completion options. *Figure 3.17* shows IntelliSense in action as a new instruction is being added to code:



```
using System;

namespace MyFirstProgram
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.Re
        }
    }
}
```

string? Console.ReadLine()
Reads the next line of characters from the standard input stream.

*__Figure 3.17:__ Advanced code completion with IntelliSense*

You can finalize auto-completion with one of the following options:

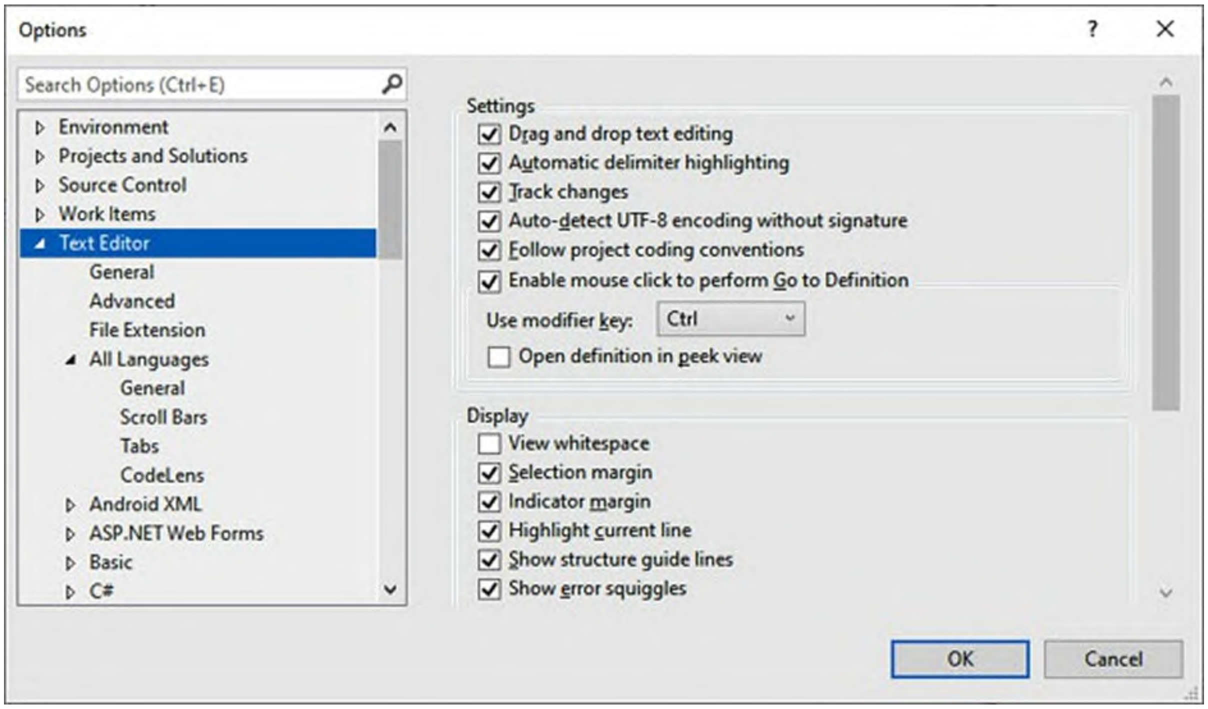
- **Tab:** Pressing *Tab* selects the highlighted auto-completion option and enables you to write other code.
- **Spacebar:** Pressing the spacebar selects the highlighted auto-completion option, adds a blank space at the end, and enables you to write other code.
- **Enter:** Pressing *Enter* selects the highlighted auto-completion option, adds a pair of empty parentheses at the end, and moves the cursor on a new line. This option is recommended when you are adding a method that does not require parameters.

- **Left parenthesis:** Pressing (selects the highlighted auto-completion option, adds a left parenthesis at the end, and enables you to supply arguments.
- **Ctrl + spacebar:** Pressing *Ctrl* + spacebar brings up the full list of IntelliSense options.

IntelliSense is activated when you type one character, and it even works with C# reserved words. When you select a word, member name, or identifier in the IntelliSense list, it will also display the documentation (if available). The list can be even filtered by member type. IntelliSense is an extremely valuable productivity tool, because provided suggestions are also based on the context. This technology is not only available in C# but to all the .NET languages and markup languages such as XAML and HTML.

## Changing the Code Editor options

The code editor can be customized in many ways. You access the code editor settings via `Tools` | `Options` and then locating the `Text Editor` node in the `Options` dialog (see *Figure 3.18*).



Settings
- ✓ Drag and drop text editing
- ✓ Automatic delimiter highlighting
- ✓ Track changes
- ✓ Auto-detect UTF-8 encoding without signature
- ✓ Follow project coding conventions
- ✓ Enable mouse click to perform Go to Definition
  - Use modifier key: Ctrl
  - ☐ Open definition in peek view

Display
- ☐ View whitespace
- ✓ Selection margin
- ✓ Indicator margin
- ✓ Highlight current line
- ✓ Show structure guide lines
- ✓ Show error squiggles

***Figure 3.18:*** *Accessing the code editor settings*

An in-depth explanation of the code editor options is out of the scope of this chapter, but here, you can find anything you can change in the settings. For example, you can enable line numbers by selecting All Languages, Line Numbers, or you can control default spacing and indentation with the Tabs node under All Languages or under the name of individual languages. **`Options`** are self-explanatory, so it is not difficult to understand how they work and what they do.

**You might be surprised to discover that the Text Editor options do not offer the possibility to customize fonts and colors. The reason is that these are located under the Environment node, Fonts, and Colors subnode. Here, you can change all the default properties of the code editor, including the colors for reserved words, identifiers, comments, and strings.**

## Compiling, running, and debugging code

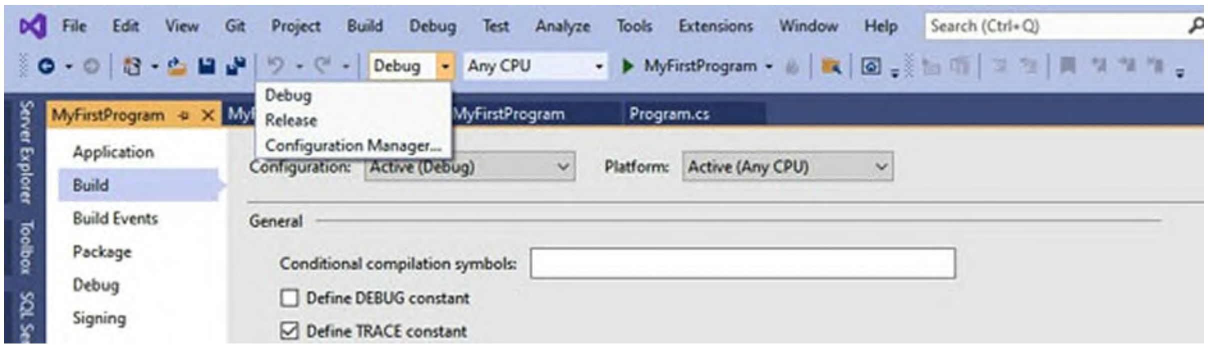
Compiling a project means generating a .NET assembly from source code and assets. In Visual Studio, this process is also referred to as building. As you learned previously in the Assemblies in .NET section of this chapter, an assembly can be a standalone application (an **`.exe`** file) or a class library (a **`.dll`** binary). However, with Visual Studio supporting many application types, including native mobile apps, the compilation process can also produce binary files for specific platforms, such as **`.apk`** files for Android and **`.ipa`** files for iOS. To compile a project into an application, you work with the Build menu. More specifically, you click on **`Build`**, **`Build ProjectName`**, where **`ProjectName`** is the name of your project. When you invoke this command, Visual Studio launches a command line tool called **`MSBuild.exe`**, which is capable of building entire solutions. When **`MSBuild`** completes building the solution, Visual Studio shows its log in the **`Output`** window. If you look back at *Figure 3.15*, you can see the output log of the build process for the **`MyFirstProgram`** sample application. The build log is very important, especially when the build process fails. In case the build process fails because of errors in the code, such errors will be listed in the **`Error List`** window (see *Figure 3.13*) described previously. If the build process succeeds, the resulting binary files will be placed in a subfolder within the **`Bin\Debug`** or **`Bin\Release`** subfolders, depending on the selected output configuration.

## Understanding configurations

By default, Visual Studio provides two configurations for compiling projects:

- **Debug**: This configuration ensures that the C# compiler generates debug symbols so that you can use the Visual Studio debugger to debug and test your applications.
- **Release**: This configuration excludes debug symbols from the compilation, and the build process is optimized for releasing your application to users.

The configuration can be set by either using the combo box located on the Visual Studio toolbar or via the **`Build`** tab of the **`Properties`** window (see *Figure 3.19*).



Configuration: Active (Debug)
Platform: Active (Any CPU)

General

Conditional compilation symbols:
☐ Define DEBUG constant
☑ Define TRACE constant

***Figure 3.19:*** *Changing the build configuration*

With Xamarin, there is an additional configuration called Ad-Hoc. It is very similar to the Release configuration, but it is optimized for distribution to testers and requires the app to be signed with the appropriate certificates. In practice, you will not use Ad-Hoc until you are ready to ship the app packages, and, at development time, you will normally work with the Debug configuration.

**Tip: Visual Studio also allows you to create custom configurations, based on the default ones. You can do this by clicking on `Build | Configuration Manager`. Creating custom configurations is not necessary in most cases, so this will not be covered here, and especially for Xamarin, it is recommended to go with the supplied configurations. So, this tip is more for your knowledge than for practical usage.**

The **`Build`** tab of the **`Properties`** window allows for customizing and controlling the compilation options. However, this tab (and the build options) changes according to the type of application you are working on. For Xamarin, the **`Build`** tab offers different options for both Android and iOS projects. For now, remember that the **`Build`** tab is the place where you can control compilation options. Especially for Android, changes on the build options will be discussed in *Chapter 5: Building Mobile Apps with Xamarin and Xamarin.Forms*.

## Running and Debugging Code

When creating applications with Visual Studio, you can run the applications from within the IDE to see how they work, and you have two options to do this:

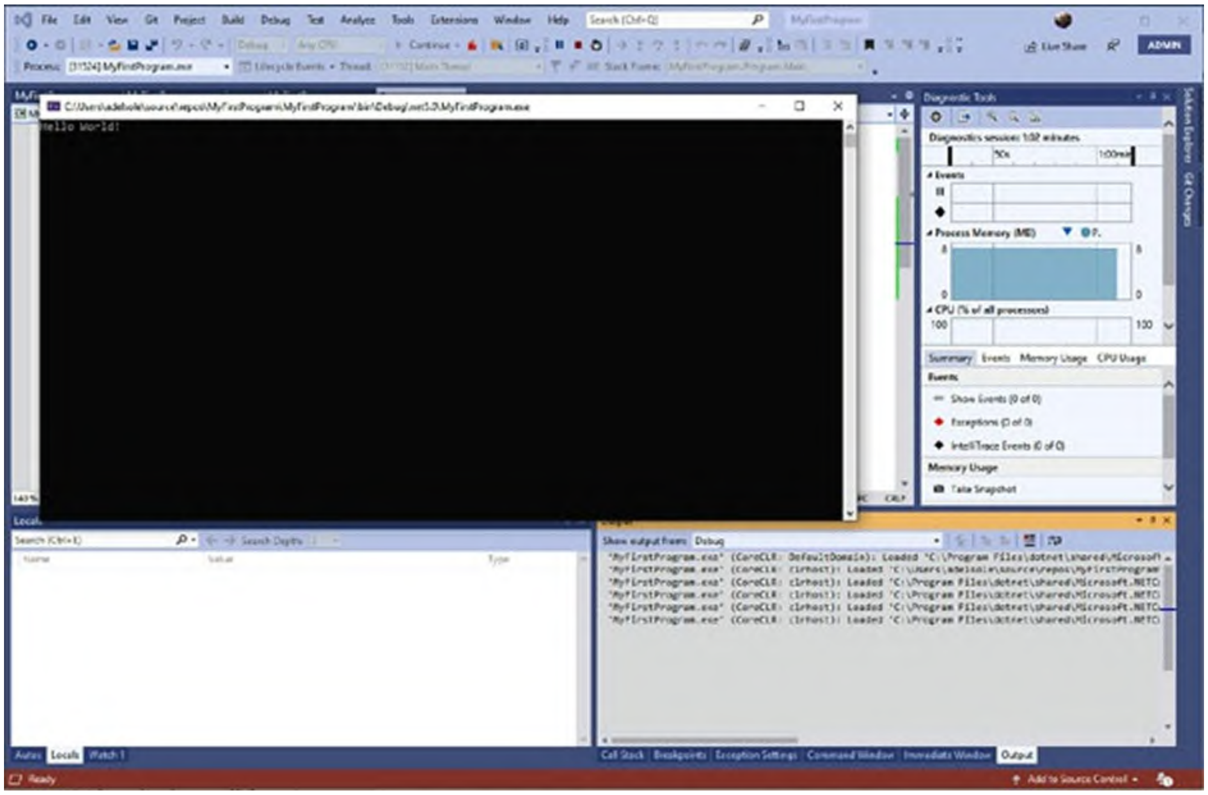
- Running the application with an instance of the debugger attached. This requires the Debug configuration to be enabled.
- Running the application without an instance of the debugger attached. It is recommended to enable the Release configuration.

**In order to start the app with the debugger attached, you can either press *F5* or click on the button in the IDE's toolbar that shows the project name with a green, Play icon. Instead, press *CTRL+F5* to start without the debugger.**

In the case of the sample Console app, the application will run in the system's console window. In the case of a Windows Forms or WPF project, the application will run within Windows and with a graphical user interface. In the case of a Xamarin project, the application will run in either a phone or tablet simulator, or a physical device. The debugger is a complex tool that analyzes the execution of every single line of code during the entire application lifetime and allows for discovering bugs, fixing errors, finding the value of variables at a certain moment in time, or simply analyzing the execution flow. Debugging is crucial in the development lifecycle, and you will spend many hours debugging your code. For this reason, it is important to understand the basic concepts of debugging applications in Visual Studio, which also apply to Xamarin development. Before going on, add the following line of code after the **`Console.WriteLine`** statement in the sample project: **`Console.ReadLine();`**. This will make the application stand by, waiting for the user input.

## Debugging your code

Debugging is the process of analyzing the execution flow of an application, finding bugs, and discovering runtime errors. To accomplish this, the debugger needs the so-called **symbols**, a set of data files that it uses to connect the source code to the runtime libraries and dependencies and analyze the flow. Visual Studio generates symbols only when you select the Debug configuration, which is, therefore, needed to debug. Now, press *F5* to start debugging. Visual Studio will first compile the project and then it will run the application. The debugger analyzes the execution of every single line of code, detects runtime errors, and allows developers to take control over the execution flow. *Figure 3.20* shows the sample application running in debug mode:



```
Hello World!
```

```
'MyFirstProgram.exe' (CoreCLR: DefaultDomain): Loaded 'C:\Program Files\dotnet\shared\Microsoft
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Users\adelsole\source\repos\MyFirstProgram
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETC
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETC
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETC
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETC
```

***Figure 3.20:*** *The application running with the debugger attached*

In the bottom area of the IDE, note that some tabs are available, such as **`Locals`**, **`Watch 1`**, **`Call Stack`**, **`Breakpoints`**, **`Command Window`**, **`Immediate Window`**, and **`Output`**. These are actually tool windows with specific debugging tools and will be recalled in the upcoming chapters where required. When an application is in debug mode, Visual Studio’s status bar and border become orange. Before you learn about the tools, you need to modify the source code of the sample application and make it intentionally cause errors that you can fix with the help of the debugger.

## Preparing the sample project for debugging

In this section, you will learn how to debug your code, how to analyze the execution flow, and how to fix errors. This requires having some code that serves to this purpose. Go to Visual Studio, and in the code, editor change the existing code as follows:

```
using System;
using System.IO;

namespace MyFirstProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      // A text message to be displayed
      string textMessage = “Hello World!";
      // Attempting to open a file that does not exist
      string textFileContent = File.ReadAllText(@"C:\MyFile.txt");
      // Displaying a text message
      Console.WriteLine(textMessage);
      // Waiting for the user input
      Console.ReadLine();
    }
  }
}
```

The next chapter will teach you more concepts about the C# programming language, but in short, here you have:

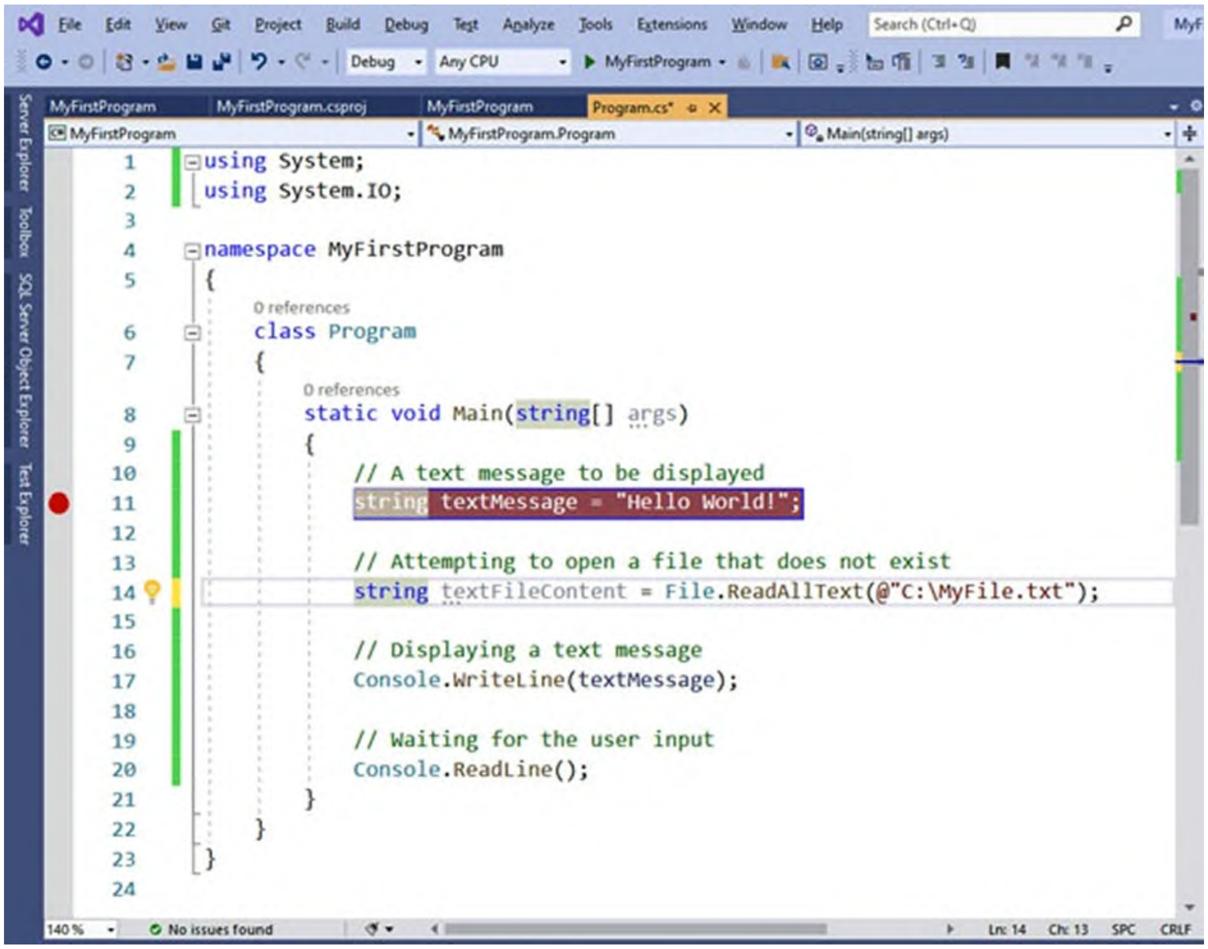
- Two **`using`** directives at the top, which simplify the way some objects are accessed in code.
- The **`root`** namespace that takes the project name.
- The definition of the **`root`** class called **`Program`**, which contains the first piece of code that will be executed when the application starts, and the **`Main`** method (in the .NET terminology, functions are referred to as methods).

- A variable of type `string` called `textMessage` that contains some text and that will be useful to analyze the execution flow.
- A variable of type `string` that contains the result of the invocation of the `ReadAllText` method from the `File` class, and that attempts to read the content of a file that does not exist. This will be useful to understand how to debug runtime errors.
- The `WriteLine` method from the `Console` class that displays the text message defined above.
- The `ReadLine` method from the `Console` class that waits for the user to press Enter and that is useful to avoid the automatic application shutdown at the end of the previous instructions.

Now that you have some source code ready, you can finally start leveraging all the power of the debugger.

## Breakpoints and data tips

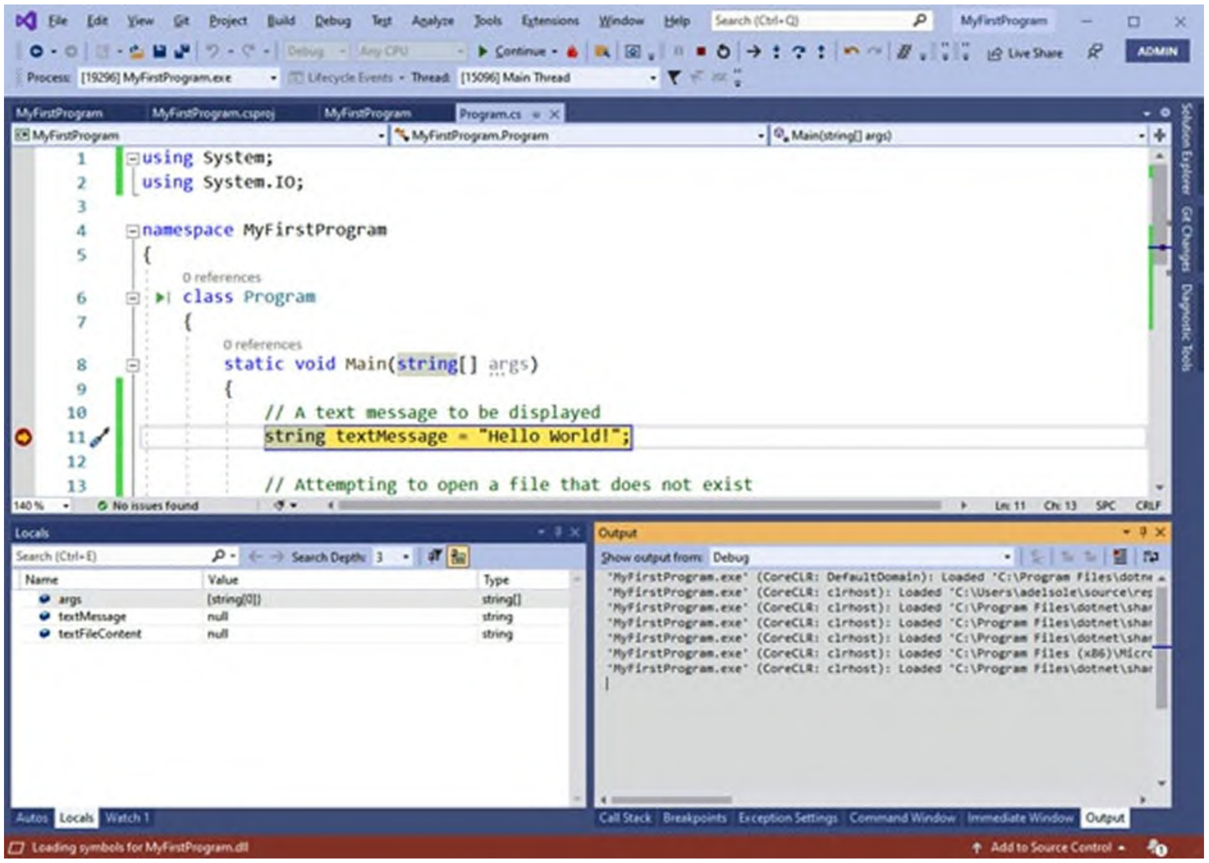
Breakpoints provide a granular way to control the execution flow of your code. You can place a breakpoint at a certain point in your code, and the execution will break there (break mode). When in break mode, you can analyze variable values and the status of your objects, taking appropriate actions. When done, you can resume the execution from the same point. You can add a breakpoint by placing the cursor on a specific line of code and then press *F9*. The line of code where you placed a breakpoint it is then highlighted in red (see *Figure 3.21*):



```
using System;
using System.IO;

namespace MyFirstProgram
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // A text message to be displayed
            string textMessage = "Hello World!";

            // Attempting to open a file that does not exist
            string textFileContent = File.ReadAllText(@"C:\MyFile.txt");

            // Displaying a text message
            Console.WriteLine(textMessage);

            // Waiting for the user input
            Console.ReadLine();
        }
    }
}
```

***Figure 3.21:*** *Adding breakpoints*

For a practical understanding of breakpoints, press *F5* to start debugging the application. When a breakpoint is encountered, the debugger breaks the execution, and the current line of code is highlighted in yellow before the line itself is executed (see *Figure 3.22*).



```
using System;
using System.IO;

namespace MyFirstProgram
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // A text message to be displayed
            string textMessage = "Hello World!";

            // Attempting to open a file that does not exist
```

| Name | Value | Type |
|---|---|---|
| args | {string[0]} | string[] |
| textMessage | null | string |
| textFileContent | null | string |

```
'MyFirstProgram.exe' (CoreCLR: DefaultDomain): Loaded 'C:\Program Files\dotne
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Users\adelsole\source\rep
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files (x86)\Micro
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
```

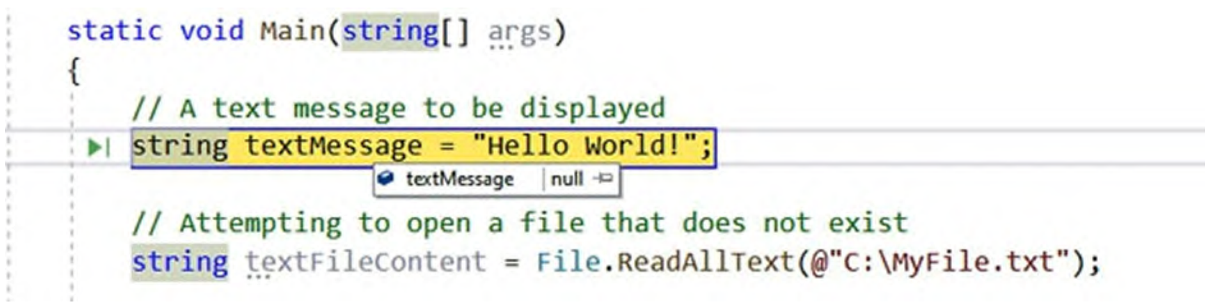
Loading symbols for MyFirstProgram.dll

***Figure 3.22:*** *Analyzing the execution flow with breakpoints*

Note how the **`Locals tool`** window shows the value of each variable at that moment. In this case, they still have no value because the first line of code has not been executed yet, but the purpose of Locals is helping you understand variables' values.

**Tip: If the tool windows you see in *Figure 3.22* are not automatically available in Visual Studio, you can manually enable them by opening the `Debug` menu and then `Windows`, selecting those of interest.**

Similarly, if you hover over the **`textMessage`** variable, you can see a tooltip displaying the content of the variable itself (see *Figure 3.23*). This feature is known as **data tips** and is extremely useful when you need to investigate the content or status of an object in a particular moment of the execution flow:



```
static void Main(string[] args)
{
    // A text message to be displayed
    string textMessage = "Hello World!";
                         textMessage | null

    // Attempting to open a file that does not exist
    string textFileContent = File.ReadAllText(@"C:\MyFile.txt");
```

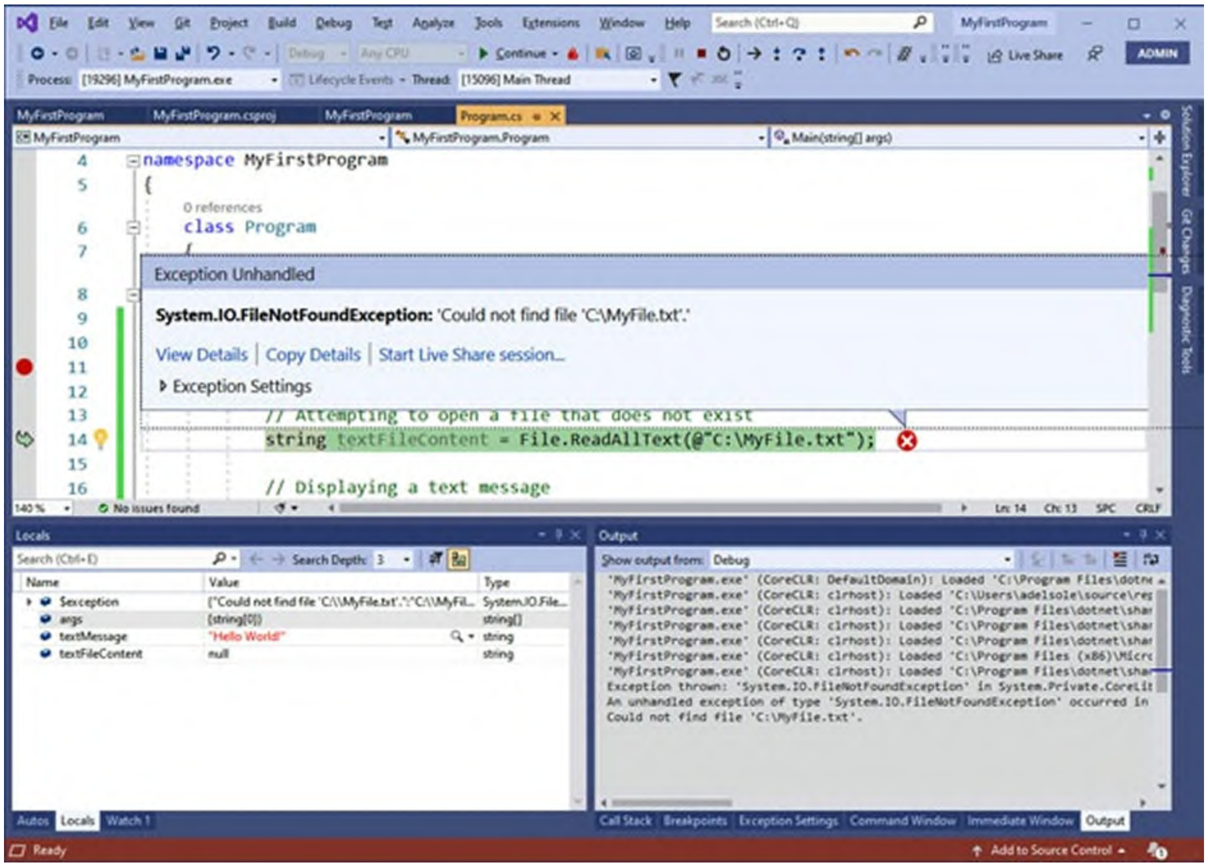
***Figure 3.23:*** *Showing variables' value with Data Tips*

Once you have completed your analysis over objects, you can either fully resume the application execution (until another breakpoint is encountered) by pressing *F5*, or you can continue by executing one statement or line of code at a time. Pressing *F10* (shortcut for the **`Step Over`** command from the **`Debug`** menu) allows you to execute a statement at a time, where statement is a set of instructions (for example, a function). With *F11*, instead, you execute a single line of code at a time. This is a shortcut for the **`Step Into`** command from the **`Debug`** menu). Continuing with the current code example, you can press *F11* to check whether the **`textMessage`** variable has been properly initialized at runtime. Whether you press *F10* or *F11*, the debugger executes the line of code where the breakpoint is placed, and Visual Studio highlights the next line in yellow. Now, you can again hover over the variable to see the assignment result, which is now **`Hello World!`**. Once investigations are completed, you can resume the execution or continue line by line.

## Detecting and fixing runtime errors

A runtime error is an error that cannot be predicted and that occurs during application execution. Usually, runtime errors are due to mistakes in the programming logic but with a valid syntax, and therefore, the compiler cannot detect them. An example of a runtime error is creating an application and giving users the ability to specify a filename, but then the file is not found on disk; another example is trying to access a database with insufficient privileges. In real-life applications, you cannot predict what the user could be doing in a specific situation, but you can predict the possibility that an error might occur. For this reason, it is your responsibility to implement appropriate error-handling logic. If you continue debugging your code, the application's execution will break again due to a runtime error. More specifically, the code is attempting to open a file that does not exist.

When a runtime error is encountered, Visual Studio breaks the execution, as shown in *Figure 3.24*:



```
namespace MyFirstProgram
{
    0 references
    class Program
    {
        ...
            // Attempting to open a file that does not exist
            string textFileContent = File.ReadAllText(@"C:\MyFile.txt");

            // Displaying a text message
```

Exception Unhandled

**System.IO.FileNotFoundException:** 'Could not find file 'C:\MyFile.txt'.'

View Details | Copy Details | Start Live Share session...

Exception Settings

Locals

| Name | Value | Type |
|---|---|---|
| $exception | {"Could not find file 'C:\\MyFile.txt'.":"C:\\MyFil... | System.IO.File... |
| args | {string[0]} | string[] |
| textMessage | "Hello World!" | string |
| textFileContent | null | string |

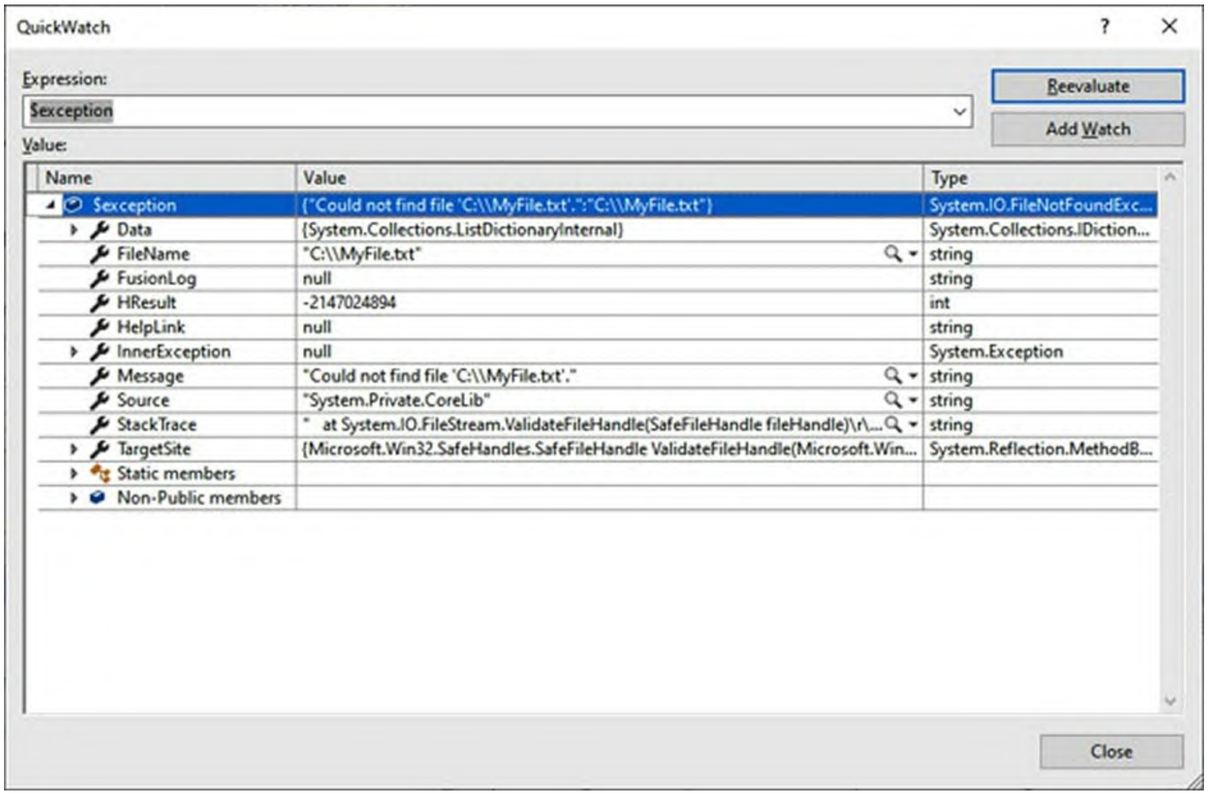
Output

Show output from: Debug

```
'MyFirstProgram.exe' (CoreCLR: DefaultDomain): Loaded 'C:\Program Files\dotne
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Users\adelsole\source\rep
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files (x86)\Micro
'MyFirstProgram.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shar
Exception thrown: 'System.IO.FileNotFoundException' in System.Private.CoreLib
An unhandled exception of type 'System.IO.FileNotFoundException' occurred in
Could not find file 'C:\MyFile.txt'.
```

***Figure 3.24:*** *Debugging runtime errors*

As you can see, the code editor highlights in light green the line of code that caused the error. A tooltip entitled **`Exception Unhandled`** also shows summary information about the error. In this case, a **`FileNotFoundException`** error was thrown and was not handled by the developer. This is the reason why the application execution was broken. The **`Exception Unhandled`** tooltip also shows a description of the error message; in this case, the description says that the specified file could not be found. At this point, you can also click on **`View Detail`**, which enables you to open the **`Quick Watch`** window, shown in *Figure 3.25*:



$exception

| Name | Value | Type |
|---|---|---|
| $exception | {"Could not find file 'C:\\MyFile.txt'.":"C:\\MyFile.txt"} | System.IO.FileNotFoundExc... |
| Data | {System.Collections.ListDictionaryInternal} | System.Collections.IDiction... |
| FileName | "C:\\MyFile.txt" | string |
| FusionLog | null | string |
| HResult | -2147024894 | int |
| HelpLink | null | string |
| InnerException | null | System.Exception |
| Message | "Could not find file 'C:\\MyFile.txt'." | string |
| Source | "System.Private.CoreLib" | string |
| StackTrace | "   at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)\r\... | string |
| TargetSite | {Microsoft.Win32.SafeHandles.SafeFileHandle ValidateFileHandle(Microsoft.Win... | System.Reflection.MethodB... |
| Static members | | |
| Non-Public members | | |

***Figure 3.25:*** *Investigating error details*

The `QuickWatch` dialog is showing a list of properties from the `FileNotFoundException` class. Among others, the `StackTrace` property is extremely useful because it shows the full hierarchy of calls to objects that caused the error. Another interesting property is `InnerException`. In the current example, it is set to null, but in many situations, it will show a hierarchy of exceptions that happened before the current one was raised. Now, what you should do is fix the error and restart the application. In the next chapter, you will learn how to implement error handling the proper way, but for now, you have discovered how to detect bugs and errors.

## Conclusion

In this chapter, you learned the basics of the .NET technology and its current status, and you started working with the Visual Studio 2019 development environment, creating your first C# project, and understanding how the debugger works. All these concepts will be useful when you will start to work with Xamarin, but first you need to get an overview of the C# programming language, which is offered in the next chapter.

## Points to remember

- .NET is an open source, cross-platform, and cross-device technology that allows for creating and running next-generation applications.
- .NET has a layered architecture that stays in between the operating system and applications.
- .NET 6 unifies all the .NET APIs into one codebase.

## Key terms

- **.NET Assembly**: Binary file resulting from the compilation process that contains executable code and metadata.
- **Base Class Library**: The main reusable library of code and components from .NET.
- **Integrated Development Environment (IDE)**: A development tool that includes code editor, debugger, and application lifecycle management tools all in one place.
- **Debugger**: Tool that allows for discovering bugs and exceptions in an application.

# CHAPTER 4

# The C# Programming Language

## Introduction

C# is a programming language you can use to write a variety of .NET applications, including mobile apps with Xamarin. It was first launched in 2002 with .NET Framework 1.0, and it was soon loved by millions of developers because it provided the ability to fully leverage the power of .NET with a syntax that was familiar for developers using C and C++, combined with the ease of usage that was typical of Visual Basic. This chapter summarizes the most relevant characteristics that you need to know as a developer writing mobile apps with Xamarin. Other, more specific features of the language will be described in the upcoming chapters, where appropriate.

**As you can imagine, it is not possible to describe in detail the C# programming language in a few pages. At the end of the chapter, you will find suggestions for books and documentation about C#.**

## Structure

In this chapter, we will cover the following topics:

- Understanding data types
- Common data operators
- Iterating objects
- Understanding loops
- Conditional code blocks
- Introducing arrays
- Object-oriented programming
- Advanced C# programming

## Objectives

After completing this chapter, you will be able to leverage the most relevant C# fundamental features that you not only need to know to build apps with Xamarin, but that you can use with any .NET project.

## Understanding data types

Applications manipulate data, which can be of different types: numbers, dates, text, and the combination of some of these into complex types that represent objects of the real life that an application must handle. .NET implements primitive data types and allows for creating custom types. In addition, .NET types are shared across languages, and this is possible to another feature of .NET called the **Common Type System**. In this section, you will learn what the Common Type System is, what primitive types are provided by .NET, and the difference between value and reference types. In the object-oriented programming section later, you will learn how to create custom types, which is something you will do very often in your daily work.

## Meet the Common Type System

.NET provides a way of manipulating data types known as **Common Type System**. This provides a unified model for exposing data types so that all the .NET languages, such as C#, Visual Basic and F#, can all consume the same data types. For example, a 32-bit integer is represented by the **`System.Int32`** data type, and all the .NET languages can invoke the **`System.Int32`** object for declaring integers because this type is provided by the .NET Framework and is language independent.

**`System` is the root .NET namespace. Namespaces can be considered as containers of types, as you will learn in the *Object-oriented programming* section later in this chapter.**

Actually, each data type is an object that inherits from the **`System.Object`** class. This class provides the primary infrastructure that all .NET types must have. .NET ships with thousands of built-in data types that all derive from **`System.Object`**, and the Common Type System ensures that all .NET types derive from it. This concept will be clearer after discussing value and reference types.

## Clarifying value types and reference types

Value types are data types that store data directly. Examples of value types are integers (**`System.Int32`**), Booleans (**`System.Boolean`**), dates (**`System.DateTime`**) and bytes (**`System.Byte`**). Value types are stored in a memory area called **stack**. The following is an example of a value type that contains a value:

```
System.Int32 anInteger = 5;
```

C# defines reserved words that represent primitive types defined in the Common Type System, such as int for **`System.Int32`**, bool for **`System.Boolean`** and byte for **`System.Byte`**, so the previous assignment can be rewritten as follows (which is also the common way of doing it):

```
int anInteger = 5;
```

**C# is case-sensitive. This means that you need to be careful about the casing of keywords and identifiers. The C# compiler will always inform you about invalid references, and IntelliSense will help you pick up the proper ones quickly, but it is something you need to keep in mind. For example, int is a reserved word, while Int is not.**

Reference types are data types that just reference the actual data. In other words, reference types store the address of their data in the stack, whereas the actual data is stored in another memory area called **managed heap**. Reference types are represented by classes, which are described in more details in the *Object-oriented programming* section. The following is an example of a reference type:

```
class Person
{
 string FirstName { get; set; }
 string LastName { get; set; }
}
```

Reference types derive from **`System.Object`** or from other classes that derive from **`System.Object`**, which defines the common infrastructure for all reference types. *Deriving* (or also *inheriting*) from an object is explained further in the Object-Oriented Programming (OOP) section later, however it means that an object automatically implements all the public members that the starting class exposes. C# also represents this with the **`Object`** keyword. Value types undergo an intermediate object called **`System.ValueType`**, which derives from **`System.Object`** and that defines the common infrastructure for all value types.

## .NET primitive types

The Base Class Library provides several built-in primitive types that you can use according to your needs. C# provides reserved words that are counterparts of the most common value type names. You can use the .NET names and the C# reserved words interchangeably. *Table 4.1* lists the most common primitive types in the .NET Framework, along with a description of each and the C# keywords:

| Type | Description | C# Keyword |
|---|---|---|
| **`System.Int16`** | Represents a numeric value with a range between –32768 and 32767. | **`short`** |
| **`System.Int32`** | Represents a numeric value with a range between –2147483648 and 2147483647. | **`int`** |
| **`System.Int64`** | Represents a numeric value with a range between –9223372036854775808 and 9223372036854775807. | **`long`** |
| **`System.Single`** | Represents a floating-point number with a range from –3.4028235E+38 to 3.4028235E+38. | **`float`** |
| **`System.Double`** | Represents a large floating-point number (double precision) with a range from –1.79769313486232e308 to 1.79769313486232e308. | **`double`** |
| **`System.Boolean`** | Accepts True or False values. | **`bool`** |
| **`System.Char`** | Represents a single Unicode character. | **`char`** |
| **`System.DateTime`** | Represents dates, times, or both in different supported formats (see the following paragraphs). | **`DateTime`** |
| **`System.Byte`** | Represents an unsigned byte, with a range from 0 to 255. | **`byte`** |
| **`System.SByte`** | Represents a signed byte, with a range from –128 to 127. | **`sbyte`** |
| **`System.UInt16`** | Represents a numeric positive value with a range between 0 and 65535. | **`ushort`** |
| **`System.UInt32`** | Represents a numeric positive value with a range between 0 and 4294967295. | **`uint`** |
| **`System.UInt64`** | Represents a numeric positive value with a range between 0 and 18446744073709551615. | **`ulong`** |

|  |  |  |
|---|---|---|
| `System.Decimal` | Represents a decimal number in financial and scientific calculations with large numbers, in a range between –79228162514264337593543950335 and 79228162514264337593543950335. | `decimal` |
| `System.TimeSpan` | Represents an interval of time, in a range between –10675199.02:48:05.4775808 and 10675199.02:48:05.4775807 ticks. |  |
| `System.Guid` | Allows the generation of globally unique identifiers. |  |
| `System.String` | Represents text. | `string` |

***Table 4.1:*** *.NET primitive data types*

## Declaring and consuming variables

Variables are symbolic names associated to a value that can potentially change. With every programming language, variables must be of a given type. When you declare variables in C#, the syntax requires you to first specify the type, then an identifier, and optionally, a value. The following example demonstrates how to declare and show the values of different variables in the Console window:

```
static void Main(string[] args)
{
  // Declares an Integer
  int anInteger = 2;
  // Declares a double and stores the result of a calculation
  double calculation = 74.6 * 834.1;
  // Declares one byte storing a hexadecimal value
  byte oneByte = 0x0;
  // Declares a string
  string sampleText = “Hello World in 2021!";
  // Declares a Boolean variable
  bool isTrueOrFalse = true;
  Console.WriteLine(anInteger);
  Console.WriteLine(calculation);
  Console.WriteLine(oneByte);
  Console.WriteLine(sampleText);
  Console.WriteLine(isTrueOrFalse);
  Console.ReadLine();
}
```

C# also allows you to declare variables without explicitly specifying their type, using the **`var`** keyword. This keyword enables a feature called local type inference and delegates to the compiler the job of identifying the type. It simply works as follows:

```
// Declares an Integer
var anInteger = 2;
// Declares a double and stores the result of a calculation
var calculation = 74.6 * 834.1;
// Declares one byte storing a hexadecimal value
var oneByte = 0x0;
// Declares a string
var sampleText = “Hello World in 2021!";
// Declares a Boolean variable
var isTrueOrFalse = true;
```

In general, it is a best practice to explicitly declare a variable type. However, there are situations in which using var is more convenient; for example, when the explicit type is extremely long or when you work with the so-called **anonymous types**.

## Clarifying the difference between value and reference types

There are differences between value types and reference types, and this is related to how they are allocated in memory and how they manage data. Value types contain the data they represent, and they are stored in a memory area called **stack**. Reference types only contain the address of the actual data they represent (a reference). The address is stored in the stack, whereas the actual data is stored in another area, called managed heap. For better understanding, consider the following lines of code that work with a value type:

```
int firstInteger = 1;
int secondInteger = firstInteger;
```

With these lines, assigning the value of **`firstInteger`** to **`secondInteger`** creates an exact copy of the value of **`firstInteger`** into the other, which also means having two integers in memory. If you change the value of **`secondInteger`**, you are not affecting the **`firstInteger`**; this can be demonstrated as follows:

```
secondInteger = 2;
Console.WriteLine(firstInteger);    // prints 1
Console.WriteLine(secondInteger);   // prints 2
```

Now, consider the following example based on a reference type. It works with the **`Person`** class defined above:

```
Person person1 = new Person();
person1.LastName = “Del Sole";
Person person2 = person1;
```

By assigning **`person1`** to **`person2`**, you are not creating a copy of **`person1`**. You are, instead, creating a copy of the address of the object. So, suppose you write the following lines:

```
person2.LastName = “White";
Console.WriteLine(person2.LastName);   // Prints White
Console.WriteLine(person1.LastName);   // Prints White
```

You will see how changing a property on an instance will consequently affect the source instance at the same address in memory. These clarifications are extremely important, and you will always need to keep them in mind, especially when working with reference types.

## Common data operators

There are different operators available in C#, such as arithmetic operators, comparison operators, and logical operators. This section provides a brief description of each category.

## Equals and Not Equals Operators

The **`==`** and **`!=`** operators allow for checking whether a condition is true or false, respectively. The following code shows an example:

```
bool trueValue = true;
if(trueValue == true)
{
  // Condition is true
}
if(trueValue != true)
{
  // Condition is false
}
```

These operators can be used wherever you need to perform a check over a condition, not only inside **`if`** blocks.

**Tip: When you need to compare two instances of the same class for equality, you will use the `Equals` method that each instance exposes, inherited from `System.Object`. There are other ways as well to compare objects for equality in .NET, but these are not discussed here.**

## Arithmetic operators

C# provides the arithmetic operators listed in *Table 4.2*.

| Operator | Description |
|---|---|
| `+` | Addition operator |
| `-` | Subtraction operator |
| `*` | Multiplication operator |
| `/` | Division operator |
| `^` | Exponential operator |
| `%` | Division remainder |

***Table 4.2:*** *Arithmetic operators in C#*

Operators' precedence follows the arithmetic rules. For example, in the following line, multiplication is executed before addition:

```
double result = 10 + 20 * 5;
```

The first four operators can also be used as increment operators. For example, suppose you had the following code:

```
int value = 1;
value++;
```

The second line increments the value of one unit. This is the so-called **postfix increment**. Now, consider the following code:

```
int value = 1;
Console.WriteLine(++value);   // Displays 2
```

In this case, the increment happens before using the value, and it is called **prefix increment**. In addition, incremental operators can be used to increment a given number of units. For example, the following line increments the value of 2 units:

```
value+=2;  // same as value = value + 2;
```

The following line multiplies the value for two:

```
a*=2;  // same as value = value * 2;
```

Remember that .NET provides the **`System.Math`** class, which exposes many methods that simplify arithmetic operations.

## Conversion operators

You will need to make conversions between .NET types very often. For example, when users tap on an item of a list in Xamarin.Forms, the selected item is represented with an object instance, but the data you are displaying is of specialized type, so you need to convert it from object to your type. You might also need to represent an integer number as a string, so you need an appropriate conversion. There are plenty of scenarios in which conversions might happen, so you need to at least have an overview of conversion operators.

## Direct type conversion

Suppose you have the following code, where **`onePerson`** is an object of type **`Person`**:

```
object aPerson = onePerson;
```

This is legal in C# but not optimal because **`aPerson`** is actually an instance of the **`Person`** class; so, you might want to convert it into the appropriate type. This can be done as follows:

```
Person result = (Person)aPerson;
```

The target type is enclosed between parentheses, and the result is assigned to the target variable. However, if the conversion fails, an error is raised. This can happen if you specify an invalid target type. You might handle the error, or you can use the as operator as follows:

```
Person result = aPerson as Person;
```

If the conversion fails, as returns null instead of raising an error. You can then check if the conversion succeeded with an **`if`** block:

```
if(result != null)
{
  // Do something
}
```

The as operator is not supported by value types unless they are nullable types (see the *Working with Nullable types* section later in the chapter).

## Converting types with the Convert class

The .NET Base Class Library offers the **`System.Convert`** class, which exposes functions for quick conversion between types. The following example demonstrates how to convert a double into a decimal:

```
double value = 2.5;
decimal result = Convert.ToDecimal(value);
```

The **`Convert`** class has methods for all the primitive types and more, such as **`ToDateTime`**, **`ToInt`**, and **`ToByte`**.

## String conversion

In C#, every type derives from **`System.Object`**, and therefore, they implement a method called **`ToString`**, which converts the current type instance into a string. The following code demonstrates how to convert a **`DateTime`** into a string:

```
string today = DateTime.Today.ToString();
```

In addition, you can still convert other types into string using the direct conversion options, as follows:

```
object aString = “Hello World";
string result = (string)aString;
string result2 = aString as string;
```

With regard to **`ToString`**, this method is flexible and powerful, and it allows for customizing the output strings with specific formatting options. You can read the official documentation at **https://docs.microsoft.com/en-us/dotnet/api/system.object.tostring**.

## Logical operators

Logical operators are special operators that enable comparisons between Boolean values and return Boolean values. There are several operators and several different scenarios, but the most common are the **`And`** and **`Or`** operators, which are represented by the **`&&`** and **`||`** literals, respectively. The following code, which includes comments, demonstrates this:

```
DateTime firstDate = new DateTime(2021, 10, 25);
DateTime secondDate = new DateTime(2021, 11, 30);
if(firstDate > DateTime.Today && firstDate < secondDate)
{
 // if firstDate is greater than today AND less than
 secondDate....
}
if (firstDate > DateTime.Today || firstDate < secondDate)
{
 // if firstDate is greater than today OR less than
 secondDate....
}
```

Another useful operator is **`!`**, which is used for negation. Look at this code:

```
if !(firstDate > DateTime.Today)
{
 // if firstDate is NOT greater than today....
}
```

With it, you can control the Boolean comparison from a negation perspective. These are the operators you will use most. Other logical operators are discussed in the official documentation (**https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators**).

## Working with strings

Strings are special types in .NET. A string is a reference type, but it is actually used as a value type. Based on this, you can assign a string to another string, as follows:

```
string hello = “Hello World!";
string target = hello;
```

Strings are immutable, which means they cannot be changed once created. This also means that when you assign or edit a string, the string object is actually creating a new string. You can compare two strings for equality or inequality using the **`==`** and **`!=`** operators, respectively, like in the following example:

```
string hello = “Hello World!";
string hello2 = “Hello World";
bool areStringEqual = hello == hello2; // false
bool areStringDifferent = hello != hello2; // true
```

More complex comparison options are supported via the **`string.Equals`** and **`string.Compare`** methods, which are not covered here. The **`string`** class exposes several methods for string manipulation. The most relevant ones are discussed in the upcoming sections.

### Checking for Null or Empty strings

In C#, strings can be null or empty. The difference is that a null string is not initialized, whereas an empty string is initialized at zero-length. In practice, you might need to check if a string is null or empty before using it. This is done via the **`IsNullOrEmpty`** method, like in the following example:

```
if(string.IsNullOrEmpty(stringToCheck))
{
  // string is null or empty
}
```

You can also use the **`IsNullOrWhiteSpace`** method to check if the string is null or only contains white spaces.

## Concatenating strings

Concatenating strings is a very common task. C# provides several ways to concatenate strings. The first way is using the addition operator, as follows:

```
string result = hello2 + hello;
```

The second way is using the **`string.Concat`** method, where each string is separated by a comma:

```
string result = string.Concat(hello2, hello);
```

The third way is known as **string interpolation** and allows for quickly embedding several variables into a string, like in the following example (note that the string must be preceded by the $ symbol):

```
string result = $"{hello2} {hello}";
```

You basically enclose variables within brackets. It is worth mentioning that you are not limited to including variables of type string; you can also include other primitive types. This is particularly useful when you need to build strings at runtime with values supplied by the user, such as URLs. The fourth way for concatenating strings is using the **`System.Text.StringBuilder`** class, as follows:

```
// Requires a using System.Text directive
string first = “first";
string second = “second";
string third = “third";
StringBuilder builder = new StringBuilder();
builder.Append(first);
builder.AppendLine(second);
builder.AppendLine(third);
string result = builder.ToString();
```

The **`StringBuilder`** class is very efficient, but it is only recommended when you have to concatenate or handle hundreds of strings. The **`Append`** method adds a string without adding a line terminator, whereas **`AppendLine`** does.

## Formatting strings

Often, you need to send output strings in a particular format, such as currency, percentage, or decimal numbers. The **`System.String.Format`** method enables you to easily format text. The following code is an example of the classic way:

```
// Returns “The cost of fuel is $1.00
Console.WriteLine(string.Format(@"The cost of fuel is {0:C}
dollars", 1));
```

```
// Returns “You are eligible for a 21.50% discount"
Console.WriteLine(string.Format(“You are eligible for a {0:P}
discount", 21.5F));
// Returns “Hex counterpart for 10 is A"
Console.WriteLine(string.Format(“Hex counterpart for 10 is
{0:X}", 10));
```

The **`Format`** method accepts a number of values to be formatted and then embedded in the main string, which are referenced with the number enclosed in brackets. For example, **`{0}`** is the second argument, **`{1}`** is the third one, and so on. Symbols specify the format; for example, **`C`** stands for currency, **`P`** stands for percentage, and **`X`** stands for hexadecimal. C# provides the symbols listed in *Table 4.3*:

| Operator | Description |
|---|---|
| **`c`** | Currency |
| **`d`** | Decimal |
| **`e`** | Scientific |
| **`f`** | Floating point |
| **`g`** | General |
| **`n`** | Number |
| **`p`** | Percentage |
| **`r`** | Hexadecimal |

***Table 4.3:*** *String formatting options*

These symbols are also accepted by the **`object.ToString`** method to format the output of any type into a string. With regard to this, you will often need to format dates via the **`DateTime.ToString`** method, but this is demonstrated later in the book.

## Iterating objects

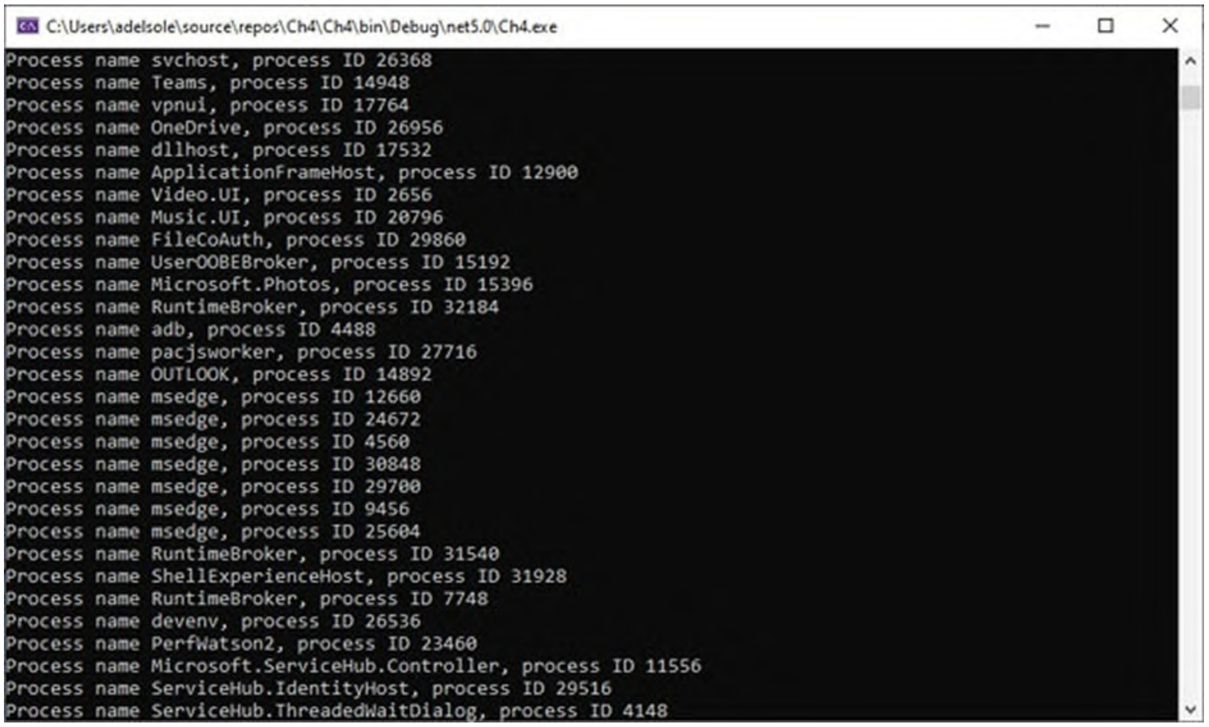
Iterations allow for executing the same action multiple times. In C#, iterations can be performed via for and foreach statements. A **`for`** statement enables you to repeat the same action (or group of actions) a finite number of times. The following code shows an example in which the same action (writing to the Console window) is repeated 10 times:

```
for (int counter = 1; counter < 11; counter++)
{
  Console.WriteLine("Action repeated {0} times", counter);
 }
```

In this kind of statements, you need to define a variable of a numeric type (counter in the preceding example) that acts as a counter. The counter is incremented, and the action is repeated until the counter is less than the specified value. Because the counter is starting from 1, repeating the action 10 times requires it to be less than **`11`** (that is, a maximum of 10). If the counter starts from 0, the counter will need to be **`< 10`**, but obviously, you will need to remember that the index of the action (0, 1 and so on) does not match the number of times the action has been done (1, 2 and so on). Note how the **`{0}`** placeholder in the string can be used to display values that are not known in advance and stored inside a variable. When you need to perform the same action over all the items in a collection or array, you can use a **`foreach`** statement. You will learn more about collections in the upcoming sections, but for now, consider this very simple example, which iterates the list of running processes on your machine, stored inside an array:

```
 // Requires a using System.Diagnostics directive
 Process[] processList = Process.GetProcesses();
 foreach(Process in processList)
 {
  Console.WriteLine("Process name {0}, process ID {1}",
  process.ProcessName, process.Id);
 }
```

*Figure 4.1* shows an excerpt of the list processes running on my machine:



```
C:\Users\adelsole\source\repos\Ch4\Ch4\bin\Debug\net5.0\Ch4.exe
Process name svchost, process ID 26368
Process name Teams, process ID 14948
Process name vpnui, process ID 17764
Process name OneDrive, process ID 26956
Process name dllhost, process ID 17532
Process name ApplicationFrameHost, process ID 12900
Process name Video.UI, process ID 2656
Process name Music.UI, process ID 20796
Process name FileCoAuth, process ID 29860
Process name UserOOBEBroker, process ID 15192
Process name Microsoft.Photos, process ID 15396
Process name RuntimeBroker, process ID 32184
Process name adb, process ID 4488
Process name pacjsworker, process ID 27716
Process name OUTLOOK, process ID 14892
Process name msedge, process ID 12660
Process name msedge, process ID 24672
Process name msedge, process ID 4560
Process name msedge, process ID 30848
Process name msedge, process ID 29700
Process name msedge, process ID 9456
Process name msedge, process ID 25604
Process name RuntimeBroker, process ID 31540
Process name ShellExperienceHost, process ID 31928
Process name RuntimeBroker, process ID 7748
Process name devenv, process ID 26536
Process name PerfWatson2, process ID 23460
Process name Microsoft.ServiceHub.Controller, process ID 11556
Process name ServiceHub.IdentityHost, process ID 29516
Process name ServiceHub.ThreadedWaitDialog, process ID 4148
```

***Figure 4.1:*** *Executing a foreach loop over the list of running processes*

Each process is represented by an instance of the **`System.Diagnostics.Process`** class. **`Process[]`** is, instead, an array of processes. If you want to retrieve some information for each process, such as the name and the identification number, you can iterate the array by using a **`foreach`** statement. You need to specify a variable (also known as a **control variable**) that is the same type as the item you are investigating. In general, **`foreach`** can be executed over objects that implement the **`IEnumerable`** interface. You can exit from a for or foreach statement at any time using the **`break;`** statement. For example, this could be the case of a condition that is met before the code block is completed.

## Understanding loops

Loops in C# allow for repeating an action until a condition is met. There are two main loops, **`do`** and **`while`**, discussed in this section.

### Introducing the do loop

The **`do`** loop executes some code while a specified Boolean expression evaluates to true. Because that expression is evaluated after each execution of the loop, a **do** loop executes one or more times. The following example demonstrates how to use a **do** loop to execute an action 10 times, and at least once:

```
int max = 0;
do
{
  Console.WriteLine(max);
  max++;
} while (max < 10);
```

Note how **while** determines the condition that evaluates to true and how it requires the condition to be enclosed between parentheses.

## Introducing the while loop

The **while** loop runs some code while a specified Boolean expression evaluates to true. Because that expression is evaluated before each execution of the loop, a **while** loop can be executed zero or more times, and this is the difference between the **while** and **do** loops. The following example demonstrates how to use a **while** loop:

```
int max = 0;
while (max < 10)
{
  Console.WriteLine(max);
  max++;
}
```

In this case, the loop will be executed at least once because **max** is assigned with **0**, which is less than **10**, and the condition is then true. But if **max** was **11**, the loop would never start.

## Conditional code blocks

Conditional code blocks allow you to execute some code only when a particular expression evaluates to a Boolean value. In C#, there are two conditional code blocks: **if** and **switch**. We will discuss both here.

## The if conditional code block

An **if** block evaluates an expression as true or false and, according to this, allows you to specify actions that should take place. The following code shows an example where the user enters a string, and its length is validated via an **if** block:

```
Console.WriteLine(“Enter a valid string (min 5 characters, max
10");
string inputString = Console.ReadLine();
if(inputString.Length < 5)
{
  Console.WriteLine(“The input string is too short");
}
else if(inputString.Length > 10)
{
  Console.WriteLine(“The input string is too long");
}
else
{
  Console.WriteLine(“The input string is valid");
}
```

The **if** condition checks whether the condition is true; if it is, the specified action is taken. You can also specify to evaluate a condition for false. You can also use an **else if** statement to provide an alternative evaluation. If no expression satisfies the condition, the **else** statement provides an action that will be executed in such a situation.

**If the action that is executed when the condition is satisfied is made of one line of code, like in the example above, it does not need to be enclosed within brackets.**

## The switch conditional code block

The **switch** statement allows you to evaluate an expression against a series of values. Generally, **switch** is used to check whether an expression matches a particular value in situations evaluated as true. If you consider the code example provided in the previous section about the **if** block, it could be rewritten using **switch** as follows:

```
switch(inputString.Length)
{
  case < 5:
    Console.WriteLine(“The input string is too short");
    break;
  case > 10:
    Console.WriteLine(“The input string is too long");
    break;
  default:
    Console.WriteLine(“The input string is valid");
    break;
}
```

In short:

- The **`switch`** condition specifies the object that must be checked.
- Each **`case`** statement checks the condition to be evaluated and allows to take actions.
- The **`default`** statement specifies an action that is executed when none of the previously checked conditions are met.
- Case and default blocks must always end with a **`break`** statement unless you use a return instruction to return a value from a function.
- Unlike general C# rules, code inside **`case`** and **`default`** statements is not enclosed within brackets.

Remember that the C# compiler is optimized to evaluate conditions in sequence, which means that the first **`case`** statement should be the one that might happen more likely to also improve performance.

## Introducing arrays

In software programming, an array is a set of elements identified by an **`index`**. Arrays in C# are zero-based index, which means that the first element has **`index = 0`**, the second element has **`index = 1`**, and so on. C# supports the following types of arrays:

- Single-dimensional arrays
- Multidimensional arrays
- Jagged arrays

Let’s start with single-dimensional arrays.

## Single-dimensional arrays

Single-dimensional arrays, as their name implies, have only one dimension and can be declared as follows:

```
 int[] arrayName = new int[10];
```

The above line declares an array of integers with a capacity of 10 elements.

The type name is followed by square parentheses. The following line demonstrates how to access an element in the array by its index:

```
int result = arrayName[2];
```

Because they are zero-based, the element with **`index = 2`** is actually the third one in the array. Obviously, the index can be represented by a variable, like in the following **`for`** loop:

```
for (int i = 0; i < arrayName.Length; i++)
{
  int result = arrayName[i];
}
```

The **`Length`** property returns the number of elements in the array.

## Multi-dimensional arrays

A multi-dimensional array allows for having items organized in rows and columns. The following code demonstrates how to create a three-dimensional array, where the first dimension can contain four items, the second can contain three items, and the third can contain two items:

```
int[,,] array1 = new int[4, 2, 3];
```

As you can see, a dimension is represented by a comma (**`,`**). The first dimension is implicit, so commas are added only starting from the second dimension. You still access elements via their index, which basically specifies the position of the element in the array, like in the following line of code:

```
int element = array1[2, 1, 1];
```

## Jagged arrays

Jagged arrays are arrays where each element in the array is another array, so they are also known as arrays of arrays. The following code declares a jagged array:

```
int[][] jaggedArray = new int[3][];
```

As you can see, the type is followed by two couples of square parentheses. In this particular declaration, the array has three elements, each a single-dimensional array. Before elements can be used, they must be initialized, like in the following example:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

**As a developer, you need to know what arrays are and how they work. However, you will use multi-dimensional arrays and jagged arrays only in specific scenarios, such as complex mathematical calculations. At least in this book, they will not be used.**

## Object-oriented programming (OOP)

In your everyday life, you perform all sorts of activities using objects. You use a glass to drink, you drive a car to reach your office, and you use a phone to connect to other people. Each of these objects has its own characteristics. For example, there are hundreds of phone models; they have different colors, different operating systems, and different carriers, but they all are phones. OOP is similar to this view of life. In fact, OOP relies on objects; for example, you can have an object that enables working on files or another object that enables managing pictures. In .NET, the development of an object is typically represented by a class. Structures are also objects, but their purpose is to represent a value more than to take actions. Objects have their own characteristics known as **properties**, but they also have some members that enable taking actions, known as methods. In this chapter, you will learn how classes in .NET development are structured and how to create your own classes, implementing all members in C#.

## Understanding access modifiers

Access modifiers set the visibility of an object and its members inside and outside of a project. They are represented by reserved words, summarized in *Table 4.4*:

| Modifier | Description |
|---|---|
| `public` | The type or member can be accessed by any other code in the same assembly or another assembly that references it. The accessibility level of public members of a type is controlled by the accessibility level of the type itself. |
| `private` | The type or member can be accessed only by code in the same class or structure. |
| `internal` | The type or member can be accessed by any code in the same assembly, but not from another assembly. |
| `protected` | |

| | |
|---|---|
| | The type or member can be accessed only by code in the same class, or in a class that is derived from that class. |
| `private protected` | The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class. |
| `protected internal` | The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly. |

***Table 4.4:*** *Access modifiers in C#*

If no modifier is specified, internal is assumed as the default. It is also worth mentioning that the availability of modifiers depends on the object or member. This will be clarified when appropriate.

## Defining reference types with classes

In .NET, classes are the types you use to define your own objects, and they are reference types. Classes are defined with a class block, as follows:

```
class Person
{
}
```

Remember that if the access modifier is not specified, C# assigns internal by default. If your classes should be consumed by other projects, for example, when you are creating a reusable library, they must be marked as public, as follows:

```
public class Person
```

Classes can define members, such as fields, properties, and methods (all covered in this chapter), and also other classes and structures. The next section covers members that can be defined within classes so that you can have a complete overview of creating your custom classes.

### Storing information with fields and properties

Fields are the places where you store information to and read information from. They are declared in the form of class-level variables and differ from local variables in that they are declared at the method or property level. The following code shows how simple it is to declare fields:

```
class Person
{
  DateTime BirthDate;
}
```

Fields can be reachable from within the class and its members, and if you specify one of the appropriate qualifiers (such as **`public`**), they can also be reached from the external world. Inside fields, you store the actual information your custom objects need. You can also provide inline initialization of fields, as in the following line:

```
DateTime BirthDate = DateTime.Now;
```

They can also be initialized in the class's constructor, and they can also be read-only. In this case, you use the **`readonly`** keyword, as follows:

```
readonly DateTime BirthDate = DateTime.Now;
```

A read-only field cannot be modified, so it needs to be initialized in-line, like in the above line, or in the class's constructor. Because fields contain the actual data the application works with, they are not the best way to provide access to the information, especially to the external world. For this purpose, you use properties. Properties are the public way that callers have to access data stored within fields. With properties, you decide the type of permissions users can have to read and write the actual information. Properties are typically used as fields, but they act as methods. The way you write properties can be either simple or extended. You need to know both ways, because in Xamarin.Forms, you will often use the extended way. Let's start from this. You write properties as follows:

```
private DateTime _birthDate;
public DateTime BirthDate
{
  get
  {
    return _birthDate;
  }
  set
  {
    _birthDate = value;
  }
}
```

In short, the get method of the property returns the value of a field, whereas the **`set`** method assigns a value to the backing field through the value reserved word. Accessing information through properties is important because you can perform validation or logic over the value in the **`set`** method. As you will see in the upcoming chapters, you will often invoke property change notification in this place. When you do not need to perform any actions on the incoming value, you can use the simplified syntax, as follows:

```
public DateTime BirthDate { get; set; }
```

This is also known as **auto-implemented properties**. When using this syntax, you do not even need to declare a backing field because the C# compiler implicitly creates one for you and automatically handles the get and set. Properties can be of any .NET type, including your own types. You use properties like you would do with any other variable, but this assumes an instance of the class. For example, you can assign a property as follows:

```
Person newPerson = new Person();
newPerson.BirthDate = new DateTime(1977, 05, 10);
Then, you can retrieve a property value like this:
DateTime dateOfBirth = newPerson.BirthDate;
Or, as another example, access the value directly:
Console.WriteLine(newPerson.BirthDate);
```

**As a general rule, you use the dot (.) to access any member of any type.**

## Running actions with methods

A method is a member that performs an operation. Methods may or may not return a value. The following are minimal examples of methods:

```
public void DoSomething()
{
 if (System.IO.File.Exists(@"C:\SomeFile.txt") == false)
  throw new System.IO.FileNotFoundException();
 else
  Console.WriteLine(“The file exists");
}
public bool DoSomethingElse()
{
 bool result = System.IO.File.Exists(@"C:\SomeFile.txt");
 return result;
}
```

The **`void`** keyword marks a method that does not return a value. For methods that return a value, the method name must be preceded by the **`return`** type. To invoke a method, you simply call its name. Continuing with the preceding example, you can invoke the **`DoSomething`** method by typing the following line:

```
DoSomething();
```

If the method returns a value, you should assign the invocation to a variable, as follows:

```
bool returnValue = DoSomethingElse();
```

This is important if you need to evaluate the value returned by the method. If you do not need to evaluate the result, you can invoke the method without assigning its result:

```
DoSomethingElse();
```

You can invoke functions anywhere you need a value. Continuing with the preceding example, the following code is acceptable:

```
Console.WriteLine(DoSomethingElse());
```

Methods exposed by a class need a dot to be invoked. If **`DoSomething`** were defined in the **`Person`** class, you would invoke it as follows:

```
newPerson.DoSomething();
```

Methods can receive parameters. In the .NET terminology, parameters are objects that methods accept and that you write when you declare a method. Arguments are the expressions you pass to a method when you invoke it, and each one satisfies a method's parameter. The following code shows a simple sample of a method definition receiving an argument and subsequent invocation of that method passing an argument; the method declaration defines a parameter called **`stringToDisplay`**, and then the method body passes an argument to the **`Console.Writeline`** method:

```
public void DisplayString(string stringToDisplay)
{
 Console.WriteLine(stringToDisplay);
}
public void SendString()
{
 DisplayString(“Xamarin for JobSeekers");
}
```

Parameters can be passed by value, which is the default, or by reference. Parameters can be passed by reference, adding the **`ref`** keyword before the parameter type (for example, **`ref int`**). The following are the differences between passing parameters by value and by reference:

- If you pass a value type by value, the compiler creates a copy of the original value, so changes made to the argument are not reflected to the original data. If you pass a value type by reference, changes made to the object referenced by the argument are reflected to the original data because, in this case, the argument is the memory address of the data.
- If you pass a reference type by reference, the compiler passes in the memory address. If you pass a reference type by value, the compiler

passes a copy of the memory pointer. In both cases, the original object will be modified, regardless of how the parameter is passed. The difference is that when you pass a reference type variable by reference, the called method can change what object the variable refers to by replacing its content with a memory pointer to a different object.

You will see some examples of parameters passed by reference later in the book. Actually, methods in .NET are even more powerful because they provide an option called **overloading**. It means providing multiple signatures of the same method, in which signature is the number and types of arguments a method can receive. The following code snippet demonstrates overloading:

```
public string FullName(string lastName, string firstName)
{
  return string.Concat(lastName, “ “, firstName);
}
public string FullName(string lastName, string firstName, int
age)
{
  return string.Concat(lastName, “ “, firstName, “ of age: “,
  age);
}
public string FullName(string lastName, string firstName,
DateTime dateOfBirth)
{
  return string.Concat(lastName, “ “, firstName, “ born on: “,
  dateOfBirth);
}
```

As you can see, there are four different implementations of one method named **`FullName`**. Each implementation differs from the others in that it receives a different number of arguments. The preceding example is simple: each implementation returns the concatenation of the supplied arguments.

## Introducing constructors and static classes

When you declare a class, you are defining a reference type. Normally, reference types cannot be used directly and need to be instantiated first. Instantiating a type allows for using one object of that type. You create an instance of a type using constructors. A constructor is a method with the same name of the class, which can optionally take parameters. Each class implicitly defines a parameterless constructor, even if you do not declare one. The following code shows an example of default, parameterless constructor:

```
public class Person
{
 public Person()
 {
 }
}
```

When you use the **new** keyword to declare a variable of a given type, you are actually invoking its constructor, like in the following line:

```
Person onePerson = new Person();
```

The constructor is also the place where you can initialize object properties and fields. If you consider the example of the **Person** class with the **BirthDate** property, you could initialize it in the constructor, as follows:

```
class Person
{
 public DateTime BirthDate { get; set; }
 public Person()
 {
  BirthDate = new DateTime(1977, 05, 10);
 }
}
```

Constructors can receive parameters so that you can initialize class's members with values determined at runtime. The following snippet shows an example:

```
class Person
{
 public DateTime BirthDate { get; set; }
 public Person(DateTime birthDate)
 {
  BirthDate = birthDate;
 }
}
```

**If you implement a constructor that receives parameters, the implicit constructor is no longer supplied by the compiler, so you need to manually provide one if you wish to be able to still create object instances without default values.**

Because constructors are methods, they also support overloading. You could, therefore, write multiple constructors, as follows:

```
public Person(DateTime birthDate)
{
 BirthDate = birthDate;
}
public Person()
{
 BirthDate = new DateTime(1977, 05, 10);
}
```

With constructors, you can have multiple instances of the same object, for example, if you need to represent multiple people, multiple books, multiple orders, multiple customers, and so on. There are situations in which you do not need to have multiple instances of an object, and a single instance is enough. In such cases, you can define a **`static`** class. The following code shows an example:

```
public static class HelperMethods
{
 public static double
  CalculatePercentage(double inputValue, double percentage)
 {
  return inputValue / 100 * percentage;
 }
}
```

Static classes are usually defined to implement general-purpose members. They do not need a constructor because there can be only one instance. Because of this, the way you invoke members of a static class requires you to write the class's name, followed by the dot and the member's name, like in the following example:

```
double result = HelperMethods.CalculatePercentage(250, 20);
```

You can also define static members inside a class that is not static. This can be useful to define members that are not connected to a specific instance of an object.

## Defining derived types with inheritance

Inheritance is an important feature in object-oriented programming. A class can inherit or derive from another class, which means that the new class can have all properties, methods, and members exposed by the first class, which is called the base class. The new class can then define its own members. Inherited members can then be overridden to adapt their behavior to the new class's context. .NET allows for single-level inheritance, which means a class can inherit from one other class at a time. Each class derives implicitly from **`System.Object`**, and the `:` symbol is used to inherit classes. The following code provides an example of a base class named **`Person`** and a derived class

named **Customer**:
```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
public class Customer: Person
{
  public string CompanyName { get; set; }
}
```
In this example, the **Customer** class exposes a new **CompanyName** property, and it exposes the **FirstName** and **LastName** properties via inheritance. There is much power with inheritance, which you will now understand with an introduction to interfaces and abstract classes.

## Understanding interfaces and abstract classes

Interfaces and abstract classes are another important pillar of OOP, and you will often use them (especially interfaces) with Xamarin. This section describes both, summarizing the key points.

### Defining interfaces

Interfaces provide a list of members that an object must implement to accomplish particular tasks in a standardized way. They are also known as contracts because they rule how an object must behave to reach some objectives. For example, .NET knows that an object that implements the **IValueConverter** interface will be used for conversion between types when working with data binding in Xamarin.Forms and will not have other targets. As another example, .NET expects you to perform databinding using objects that implement the **INotifyPropertyChanged** interface, and that are, therefore, capable of sending a change notification. An interface is a reference type defined within an interface **{ }** block. Interfaces define only signatures for members that classes will then expose and are a set of the members' definitions. Imagine you want to create an interface that defines members for objects that represent different categories of people. This is accomplished with the following code:
```
public interface IPerson
{
  string LastName { get; set; }
  string FirstName { get; set; }
  DateTime DateOfBirth { get; set; }
  string FullName();
}
```

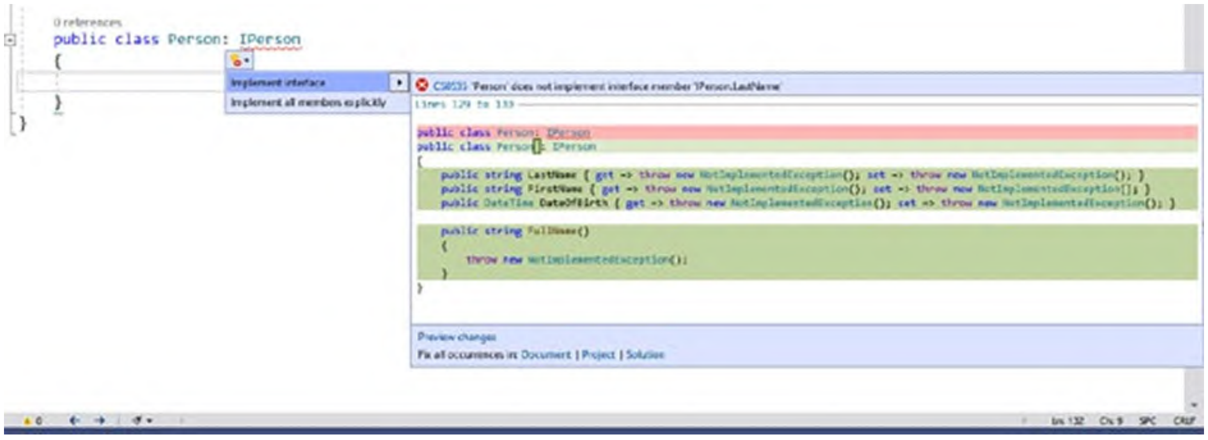
By convention, interface identifiers begin with a capital **`I`**. Though not mandatory, this is strongly recommended. The interface definition contains only members' definitions with no body. For **`FullName`** method definitions, there is only a signature but not the method body and implementation, which are left to classes that implement the interface. Members defined within interfaces cannot be marked with one of the access modifiers, which are public by default.

## Implementing interfaces

Implementing interfaces means telling a class that it needs to expose all members defined within the interface. You do this by using the **`:`** symbol, followed by the name of the interface. IntelliSense will offer a list of available interfaces and, when you select **`IPerson`**, the code editor will underline it with red squiggles because the implementation is still missing. However, you can quickly provide a basic implementation, as follows:

- Hover over **`IPerson`**.
- Click on the light bulb icon.
- Click on the **`Implement Interface`** option.

*Figure 4.2* demonstrates how to accomplish this:



```
0 references
public class Person: IPerson
{

}
}
```

CS0535 'Person' does not implement interface member 'IPerson.LastName'

Lines 129 to 133

```
public class Person: IPerson
public class Person : IPerson
{
    public string LastName { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    public string FirstName { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    public DateTime DateOfBirth { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

    public string FullName()
    {
        throw new NotImplementedException();
    }
}
```

***Figure 4.2:*** *Quickly implementing an interface*

Not only can you see a preview, but Visual Studio will provide a basic implementation for you, which needs to be rewritten though. The following code snippet shows how to implement the **`IPerson`** interface within a **`Person`** class:

```
public class Person : IPerson
{
 public string LastName { get; set; }
 public string FirstName { get; set; }
 public DateTime DateOfBirth { get; set; }
 public string FullName()
 {
  return $"{LastName} {FirstName}";
 }
}
```

Apart from being used as contracts, interfaces are important for a programming pattern known as polymorphism. Because multiple classes can implement the **`IPerson`** interface, you could declare the following variable that is able to receive any objects that implement **`IPerson`** and manipulate members that are common to all the instances:

```
IPerson person = newPerson;
Console.WriteLine(person.FullName());
```

The **`newPerson`** object could be of type **`Person`** or any other type that implements **`IPerson`** and, through polymorphism, you can manipulate common members, regardless of the original type.

## Introducing abstract classes

Abstract classes allow for creating a base implementation for derived objects but cannot be used directly. Consider the following **`Person`** class:

```
public abstract class Person
{
 public string LastName { get; set; }
 public string FirstName { get; set; }
 public virtual string FullName()
 {
  return $"{LastName} {FirstName}";
 }
}
```

It provides an infrastructure that can be common to any class that represents people in a more specialized way, such as customers, contacts, and friends. It is marked as abstract, so you cannot create an instance of the class, and it must be inherited. Also, note how the **`FullName`** provides a basic implementation, but it is marked as virtual, which gives the option to redefine its behavior. Following is an example of a **`Customer`** class that derives from **`Person`** and that overrides the **`FullName`** method:

```
public class Customer: Person
{
 public int CustomerID { get; set; }
 public override string FullName()
 {
  return $"Customer ID: {CustomerID}, Last name: {LastName}";
 }
}
```

You are not obliged to override a **`virtual`** method, and you can leverage the original implementation. In this case, you use the **`base`** keyword, followed by the member's name.

**In summary, interfaces establish which members an object must implement in order to satisfy a contract; abstract classes establish which members a derived object must implement, with the option to provide a base implementation, and without satisfying any contract.**

The opposite of abstract classes is sealed classes. A class marked with the sealed keyword cannot be inherited. This can be useful if you build libraries that are sold to third parties, and you do not want new objects to be created starting from yours.

## Organizing types within namespaces

Namespaces can be considered as containers of types. Namespaces are defined within namespace **`{ }`** blocks. Every namespace can expose the following types and members:

- Classes
- Structures
- Interfaces
- Enumerations
- Delegates
- Nested namespaces

The following listing provides an example of namespace defining some of the

aforementioned types:
```
namespace People
{
 public interface IContactable
 {
  bool HasEmailAddress { get; }
 }
 public abstract class Person
 {
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public override string ToString()
  {
   return FirstName + “ “ + LastName;
  }
 }
 public enum PersonType
 {
  Work = 0,
  Personal = 1
 }
 public class Contact: Person, IContactable
 {
  public string EmailAddress { get; set; }
  public override string ToString()
  {
   return base.ToString();
  }
  public bool HasEmailAddress
  {
   get
   {
    if (string.IsNullOrEmpty(this.EmailAddress))
     return false;
    else
     return true;
   }
  }
 }
 public struct PersonInformation
 {
  public PersonType PersonCategory { get; set; }
  public bool HasEmailAddress { get; set; }
 }
}
```
The purpose of namespaces is better organizing types. In some situations, an

object's hierarchy could expose two different types with different behaviors, but with the same name. For example, imagine you have two **`Person`** classes; the first one should represent business contacts, and the second one should represent your friends. Of course, you cannot create two classes with the same name within one namespace. Because of this, you can organize such types in different namespaces, and thus, avoid conflicts.

**By convention, a namespace's name should have the following form: `CompanyName.ProductName.NamespaceName`. This is not mandatory, and it's completely up to you to follow this convention. If you use third-party libraries, namespaces are defined in this way.**

## Accessing types within namespaces with using directives

You will often need to invoke types defined within long-named namespaces. To invoke types, you need to write the full name of the type, which includes the identifier of the namespace that defines a particular type, as in the following code:

```
System.IO.FileStream textFile =
  new System.IO.FileStream(@"C:\test.txt",
  System.IO.FileMode.Open);
```

Accessing an object by typing the full name, including the namespace, is known in .NET as fully qualified name. To write simpler and cleaner code faster, you can add using directives to shorten the way you access types. For example, at the top of your code file, you can add the following directive:

```
using System.IO;
```

This lets the compiler know that it needs to look for types defined in the **`System.IO namespace`**. Then, you can rewrite the previous code snippet as follows:

```
FileStream textFile = new FileStream(@"C:\test.txt",
FileMode.Open);
```

You will still use the fully qualified name if you access two objects with the same name but from two different namespaces.

## Defining value types with structures

Structures allow for creating custom value types. You find a lot of similarities between classes and structures, although this section explains some important differences. You create structures using a structure `{ }` block. The following code provides an example that represents a point coordinate with the timestamp it was created:

```
public struct ExtendedCoordinate
{
  public double X { get; set; }
  public double Y { get; set; }
  public DateTime TimeStamp { get; set; }
  public ExtendedCoordinate(double x, double y)
  {
    X = x;
    Y = y;
    TimeStamp = DateTime.Now;
  }
}
```

Structure can expose several members, such as fields, properties, methods, and even events as it happens for classes. However, structures should really be used only when you need types that need to create a copy of the data they handle, as explained below.

**Structures do not support inheritance. They only inherit from `System.ValueType`, but no further level is allowed.**

## Assigning structures to variables

Structures are value types, so assigning an instance of a structure to a variable declared as of that type creates a full copy of the data. The following code demonstrates this:

```
ExtendedCoordinate coordinate = new ExtendedCoordinate(45.2,
37.1);
```

This is one of the biggest and most important differences between structures and classes. Obviously, structures can also be passed as method parameters.

## Visibility of structures and their members

Structures and their members have access modifiers as well as classes. Structures only accept private, public, and internal qualifiers. If no qualifier is specified, internal is provided by default. Structures' members can be qualified with modifiers described in *Table 4.2*.

## Defining enumerations

Enumerations are another kind of value type available in .NET. They represent a group of constants enclosed within an **`enum { }`** code block. An enumeration derives from **`System.Enum`**, which derives from **`System.ValueType`**. The following is an example of enumeration:

```
public enum Food
{
 Bread,      // 0
 Tomato,     // 1
 Potato,     // 2
 Chicken,    // 3
 Fish        // 4
}
```

By default, enumerations are sets of integer values. The preceding code defines a Food enumeration of type **`int`**, which stores a set of integer constants. The C# compiler can also automatically assign an integer value to each member within an enumeration, starting from zero, as indicated in comments. You could manually assign custom values, but you should avoid this when possible because the standard behavior ensures that other types can use your enumeration with no errors.

## Using enumerations

You use enumerations as any other .NET type. For example, consider the following method that receives the **`Food`** enumeration a parameter and displays a response depending on which value has been passed:

```
private void CheckFood(Food foodList)
{
 switch (foodList)
 {
  case Food.Bread:
   {
    Console.WriteLine(“You chose Bread");
    break;
   }
  case Food.Chicken:
   {
    Console.WriteLine(“You chose Chicken");
    break;
   }
  default:
   {
        Console.WriteLine(“All food is good");
        break;
      }
  }
}
```

The following code snippet then declares a variable of type **`Food`**, assigns a value, and invokes the method by passing the variable:

```
var myFood = Food.Bread;
CheckFood(myFood);
```

Note how IntelliSense comes in when you need to specify a value whose type is an enumeration.

## Implementing error handling

As you know, one of the characteristics of .NET languages is interoperability, which means one language must understand and use code written in other language. This is why there is a need for a common way for handling errors. In .NET, errors are identified as exceptions. An exception is an instance of the **`System.Exception`** class (or of a specialized class derived from it) and provides deep information on the error that occurred. You perform exception handling by writing a **`try..catch..finally`** code block. The logic of the flow behind this block is the following sequence:

- Try to execute the code.
- If you encounter an exception, take the specified actions.
- Whenever the code execution succeeds or it fails due to an exception, execute the final code.

The following example demonstrates how to control the exception flow in case some code attempts to open a file and an error occurs:

```
// Requires a using System.IO; directive
Console.WriteLine(“Specify a file name:");
string fileName = Console.ReadLine();
FileStream myFile = null;
try
{
  myFile = new FileStream(fileName, FileMode.Open);
  // Seek a specific position in the file.
  // Just for example
  myFile.Seek(5, SeekOrigin.Begin);
}
```

```
catch (FileNotFoundException ex)
{
 Console.WriteLine($"File {ex.FileName} not found.");
}
catch (Exception ex)
{
 Console.WriteLine($"An unidentified error occurred:
 {ex.Message}");
}
finally
{
 myFile?.Close();
}
```

When you use **`catch`** to intercept an exception, you can pass a variable that stores the instance of the exception object to retrieve a lot of useful information. With regard to the previous code, it attempts to open a file and search for a specific position within it. If the file is not found, the **`FileStream.Seek`** method throws a **`FileNotFoundException`**. Among other things, this exposes a **`FileName`** property that contains the name of the file that was not found. In general, all the exception classes expose a property called **`Message`**, which contains the error message, and a **`StackTrace`** property, which is useful for developers to understand which part of the code caused the error. The Base Class Library, and all third-party libraries in general, include a large number of specialized exceptions, and it is a best practice to catch all those that can happen in a specific scenario. In the code snippet above, a specialized exception is caught for files that are not found, but because other unpredictable errors can happen, a generic exception is also caught. The **`finally`** block is completely optional, but it is very important in some cases. In fact, it allows for executing code regardless of the success or failure of the **`try`** block. If you have open files, like in the preceding example, or open database connections, this is the place where you can close all the existing references.

## Implementing events

Events are a way to notify other objects that something has happened. An event is declared using the **`event`** keyword. When you define an event, you also need a class that contains the information that the event wants to share with callers. .NET defines a base class called **`EventArgs`** and that you can inherit to implement your own. Suppose you want to notify callers about the completion of reading a text file. You can implement an event and an event arguments class, as follows:

```
class EventsDemo
{
 public event EventHandler<OperationCompletedEventArgs>
 FileCompleted;
}
class OperationCompletedEventArgs: EventArgs
{
 public string FileName { get; set; }
 public OperationCompletedEventArgs(string fileName)
 {
  FileName = fileName;
 }
}
```

The following are the key points:

- **`EventHandler`** is a delegate, which is a special .NET type that can be thought of as a function pointer.
- The **`OperationCompletedEventArgs`** exposes the information you want to share with event subscribers, in this case, the filename, which must be supplied with the constructor.

An event is raised by calling its **`Invoke`** method as follows:

```
public void OpenFile(string fileName)
{
 string result = System.IO.File.ReadAllText(fileName);
 var eventArgs = new OperationCompletedEventArgs(fileName);
 FileCompleted?.Invoke(this, eventArgs);
}
```

You first create an instance of the event arguments class supplying the required information, and then you call the Invoke method of the event. There are two additional points to highlight here:

- The **`Invoke`** method invocation is preceded by the **`?`**. symbol. As a general rule, in C# this ensures that, if the object is null, no exception is **`thrown`** and null is returned to the caller.
- This is the first time you encounter the **`this`** keyword. It represents the current instance of the class.

Objects that want to be notified of an event must subscribe for it. The following code demonstrates this:

```
var demo = new EventsDemo();
demo.FileCompleted += Demo_FileCompleted;
demo.OpenFile(@"c:\myfile.txt");
```

You subscribe to an event with the += operator, followed by the name of a method, known as event handler, that will be executed when the event is intercepted. After you type **+=**, you can press Tab twice, and Visual Studio will generate an event handler for you. Obviously, the body of the event handler must be replaced with your code. The event handler signature has two parameters:

- The object that is sending the event.
- An instance of the event arguments class that contains the event information.

The following is a simple example that demonstrates how the event handler is made and how to read its information:

```
private void Demo_FileCompleted(object sender,
OperationCompletedEventArgs e)
{
  string fileName = e.FileName;
}
```

Events are commonly used when handling actions on the user interface, so you will see additional examples in the upcoming chapters.

**This chapter could only make a mention about delegates, but actually, they are an important part of the .NET architecture. The official Microsoft documentation has a dedicated page at https://docs.microsoft.com/en-us/dotnet/csharp/delegates-overview.**

## Advanced C# programming

This section describes advanced concepts and techniques that you will use frequently in your daily life as a developer working with Xamarin. As you can imagine, like for the other topics, there is much more than what can be summarized in this chapter, so it is always recommended that you read a specific book about C# programming if you do not have enough experience or if you feel that you need further explanation.

## Generics and Nullable Types

You will often use generics and nullable types when writing code for Xamarin. Generics are .NET types that can adapt their behavior to different types of objects from a single implementation. They can hold only the specified type and avoid accidents of handling objects of different types. There are dozens of generic types defined in .NET, and you can create your own generics (not covered in this book), but the most frequent situation for you is using generic collections. For example, .NET defines the **`System.Collections.Generics.List<t>`** class, an object that can hold a list of objects of only one type. The **`<t>`** literal represents the generic implementation and is called **type parameter**. Another collection that you will use very frequently in Xamarin.Forms is the **`System.Collections.ObjectModel.ObservableCollection<t>`**, which derives from **`List<t>`** and is able to notify the user interface of changes inside the elements it contains. The following snippet demonstrates how to create a list of integers, and the technique is the same for every type, including custom classes:

```
var integerList = new List<int>();
integerList.Add(1);
integerList.Add(2);
Console.WriteLine(integerList[1]); // writes 2
integerList.Remove(1); // removes 1
```

You first need an instance of the generic type, and then you can invoke the **`Add`** method to add new objects of the specified type. You can access individual elements in the list using their index, which is zero-based. You can also remove members via the **`Remove`** method, which removes the specified value or instance (not the index). Another way to create a list with generics is using a feature known as **collection initializers**, which is useful when the values or instances you need are represented by variables, like in the following code:

```
int first = 0;
int second = 1;
var integerList = new List<int> { first, second };
```

With this syntax, the list is created and populated with the series of values enclosed between brackets. When you implement your own generics, you can also provide some constraints. For example, you might want to decide that a generic can only work with reference types that implement a specific interface. Generics is a much more complex topic, so I recommend that you read the official documentation at **https://docs.microsoft.com/en-**

**us/dotnet/csharp/fundamentals/types/generics**.

## Working with Nullable types

All .NET types have a default value. For reference types, it is null. Value types have a default value depending on the type itself; for example, it is zero for int or false for **`bool`**. Value types cannot be null. However, there are situations where it would be important for value types to also support null. This is the case of mapping database columns to .NET objects or having three-state objects (such as true, false, null). In C#, this can be accomplished with nullable types, which are generic types. You have two ways of declaring nullable types:

```
Nullable<int> oneInteger = 0;
int? oneInteger = 0;
```

The **`Nullable<t>`** generic type adds support for null values to any value type, and you can also use a simplified syntax, adding a question mark after the declared value type. The following code shows an example of how you consume nullable types:

```
if (oneInteger.HasValue)
  Console.WriteLine(oneInteger.Value);
```

While you can assign a nullable type directly with a value, you must first check if the object is not null. This can be done by evaluating the **`HasValue`** property, which returns true if the object is not null. At this point, you can access the actual value via the **`Value`** property. Attempting to access a nullable type that is null will result in a **`NullReferenceException`**.

## Language INtegrated Query (LINQ)

**Language INtegrated Query** (**LINQ**) is a feature that allows querying collections of objects using a syntax that recalls the SQL language. This is one of the most complex and powerful features in C#, so here, it is only possible to make an introduction with a focus on the way of usage that will be useful in the later chapters. Suppose you have a **`List<Person>`** called people and you want to retrieve a subset of people for whom the year of birth is less than 2000. With LINQ, you could write this code as follows:

```
var query = from person in people
      where person.DateOfBirth.Year < 2000
      select person;
```

The **`from`** clause allows for referencing every single instance in the collection, where provides an option to add filters to the query via conditions and select allows for adding every instance that matches the condition to a new set of data. The result of the query is an **`IEnumerable<t>`** object (which changes to **`IQueryable<t>`** if you are working with a data framework like **`Entity`** Framework). In general, LINQ can be used against every object that implements the **`IEnumerable<t>`** interface. Obviously, there are many more possible options and clauses, but this is the basic syntax that you need to know. In fact, you can group query results, perform mathematical aggregations, and so on. You can also retrieve individual elements from a query. For instance, you can retrieve the first or last element in the collection. The following example shows how to retrieve the first element of a sequence:

```
var query = (from person in people
     where person.DateOfBirth.Year < 2000
     select person).First();
```

**`First`** is an extension method that returns one instance of the type it is invoked on; so now, **`query`** will be of type **`Person`**. .NET is full of extension methods, and it is possible to create custom ones. As the name implies, extension methods extend an existing object with new functionalities, without the need to have the source code of the original object. Actually, extension methods can be used in dozens of different scenarios, but probably, the most common is with lambda expressions. A lambda expression can be considered as anonymous method that allows for performing queries and evaluating complex expressions in one line. For instance, with lambda expressions you could rewrite the previous LINQ query as follows:

```
var query = people.Where(person => person.BirthDate.Year <
2000).First();
```

The **`=>`** operator represents the syntax of a lambda expression and provides a way to declare a reference variable to each element in the sequence. Note how **`Where`** is also an extension method to filter the results. You can combine multiple conditions using the **`&&`** and **`||`** operators. The First method throws an exception if no element that matches the condition is found. If you want to avoid an exception, you can use **`FirstOrDefault`**, which returns the default value for the current type (null in this case). Similarly, LINQ provides the **`Last`** and **`LastOrDefault`** methods to retrieve the last element in a sequence. Everything that can be done with the regular LINQ syntax can be done with lambda expressions, and you can perform more complex operations in-line. You will see more examples in the upcoming chapters.

**LINQ is not only one of the most powerful features of .NET languages, but also one of the most complex, and it is impossible to describe all of its potential in a few pages. It is strongly recommended that you take a look at the Microsoft documentation (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/).**

## Asynchronous programming

Normally, code runs synchronously, which means one operation at a time. With short-running operations, there is no problem. With long-running operations, an application can become unresponsive and can freeze until the operation completes, and the user will certainly get nervous. As a general programming concept, it is a best practice to run long-running operations on a separate thread. A thread can be thought of as an individual unit of work. The user interface runs on a specific thread, so if all the long-running code affects the user interface thread, the application will freeze. This is why delegating potentially long-running operations to a separate thread makes the app more responsive; this technique is known as multithreading. However, dealing with threads can introduce other problems, such as (but not limited to) the following:

- An object handled by a thread cannot be accessed by another thread, and to make it possible, you have to deal with very complex programming techniques.
- Access to resources must be implemented in a way that they are locked to other threads until a first thread completes its work.

Everything can be solved, of course, but it requires complex work. In order to make things simpler while still keeping the application responsive, .NET provides the asynchronous programming pattern since 2012. In short, an individual thread can be divided into multiple tasks, where each task can work asynchronously, without creating a separate thread. These are the relevant points:

- The **`async`** modifier is added to method declarations.
- An **`await`** operator is used within **`async`** methods to wait for a task to complete without blocking the thread.

To understand how it works, consider the scenario of reading a long file. The longer the file, the higher the risk of blocking the user interface until the operation completes. The following example shows how to create a method that reads a file asynchronously:

```
private async Task<bool> ReadFileAsync(string fileName)
{
  try
  {
    string fileContent = await File.ReadAllTextAsync(fileName);
    Console.WriteLine(fileContent);
    return true;
  }
  catch (Exception)
  {
    return false;
  }
}
```

Here are a few additional points of interest:

- By convention, the name of asynchronous methods should end with the async suffix.
- Asynchronous methods return objects of type **`System.Threading.Tasks.Task`**, with no return value (that is, **`void`**), or **`System.Threading.Tasks.Task<T>`** when they return a value. However, the **`return`** statement just takes the original type, not **`Task<t>`** (see the preceding code).
- The compiler creates a unit of work (**`task`**) inside the same thread, which keeps the application responsive and solves all the problems related to having multiple threads.

Asynchronous programming is probably the most complex topic in C# and should never be used everywhere; it should only be used where long-running operations might block the user interface. In this section, you learned the most basic concepts, which are also important with Xamarin because you will use this pattern all the time. However, if not used correctly, they can lead to inefficient code. For this reason, do not forget to read the official documentation (**https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/**).

## Conclusion

C# is a high-level, compiled, and OOP language that you can use to build a variety of applications, from desktop to the web and mobile, passing through cloud. Microsoft has, without a doubt, always considered C# as the most important .NET language and has continuously invested on it. C# is also the programming language you will use to build mobile applications with Xamarin, starting from the next chapter. This chapter could only summarize the most important features of the language that you will need to keep in mind as a mobile app developer, but it is certainly necessary for you to go through additional documents, as suggested in the final subsection. Now, sit down, fasten your seatbelt, and get ready to start your journey as a Xamarin developer.

## Suggested readings

- Book: *Mastering C# 8.0*, BPB Publishing (**https://bpbonline.com/collections/programming/products/mastering-c-8-0-book-ebook**).
- Microsoft official C# documentation (**https://docs.microsoft.com/en-us/dotnet/csharp/**).

# CHAPTER 5

# Building Apps with Xamarin and Xamarin.Forms

## Introduction

After learning the basics of Microsoft Visual Studio and the C# programming language, it is time to start working on mobile app development with Xamarin. Though this book focuses on Xamarin.Forms, it is important for you to know how native projects work. The reason is that a Xamarin.Forms solution relies on native projects, so you must know how they are structured and how you can configure them. The first part of this chapter explains how to create, configure, and debug native Xamarin.Android and Xamarin.iOS projects. In the second part, you will learn how to create a Xamarin.Forms solution, discussing its structure and understanding the fundamentals of this platform, which is required for the later chapters.

## Structure

In this chapter, we will cover the following topics:

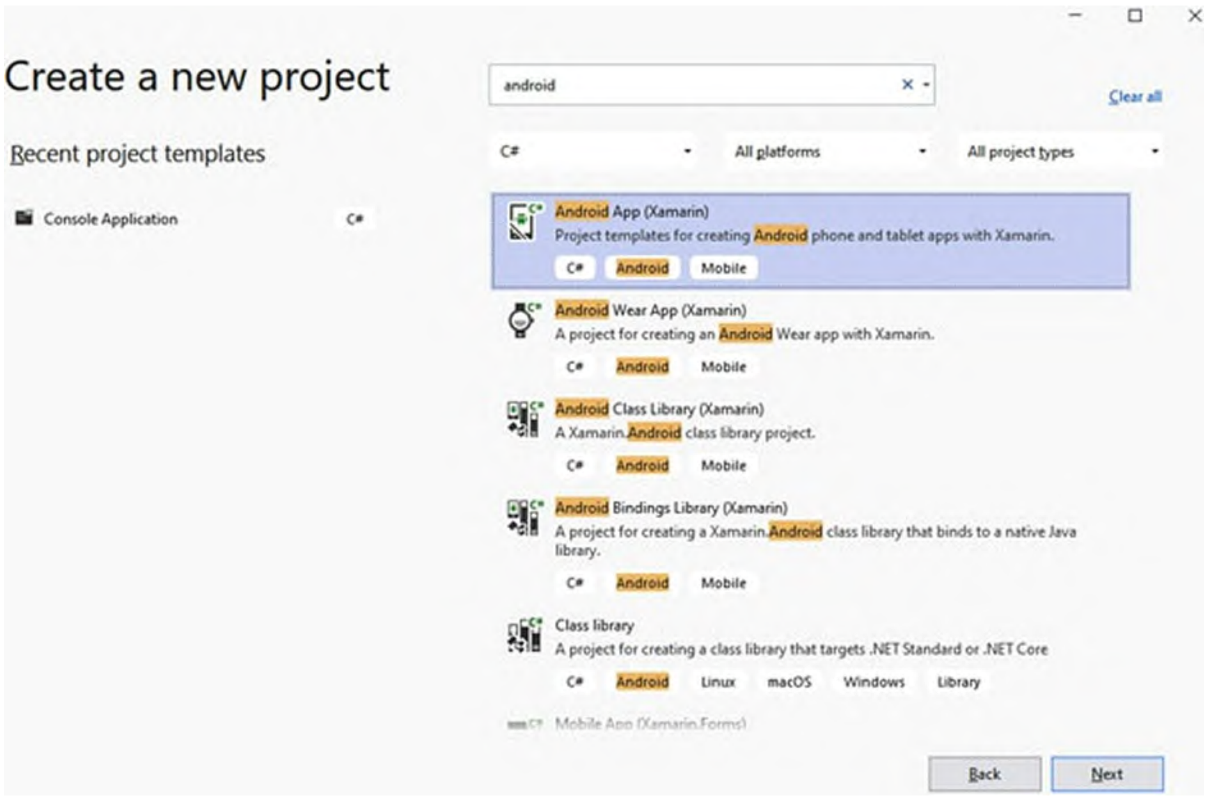
- Understanding Xamarin.Android projects
- Understanding Xamarin.iOS projects
- Cross-platform projects with Xamarin.Forms
- Preparing apps for publication

## Objective

After completing this chapter, you will be able to create Xamarin.Forms solutions and able be able to configure the backing native projects. You will also learn the most important concepts about preparing apps for distribution.

## Understanding Xamarin.Android Projects

By reusing the skills learned in the previous chapters, open Visual Studio and go to the `Create a new project` dialog (or select `File` | `New` | `Project` if you were already in the IDE). The project template you need to create a Xamarin.Android project is called **Android App** (**Xamarin**). *Figure 5.1* shows how it appears (you can also filter the list to find it quickly, like in the figure).



Create a new project

Recent project templates

Console Application C#

android

C# · All platforms · All project types

Android App (Xamarin)
Project templates for creating Android phone and tablet apps with Xamarin.
C# Android Mobile

Android Wear App (Xamarin)
A project for creating an Android Wear app with Xamarin.
C# Android Mobile

Android Class Library (Xamarin)
A Xamarin.Android class library project.
C# Android Mobile

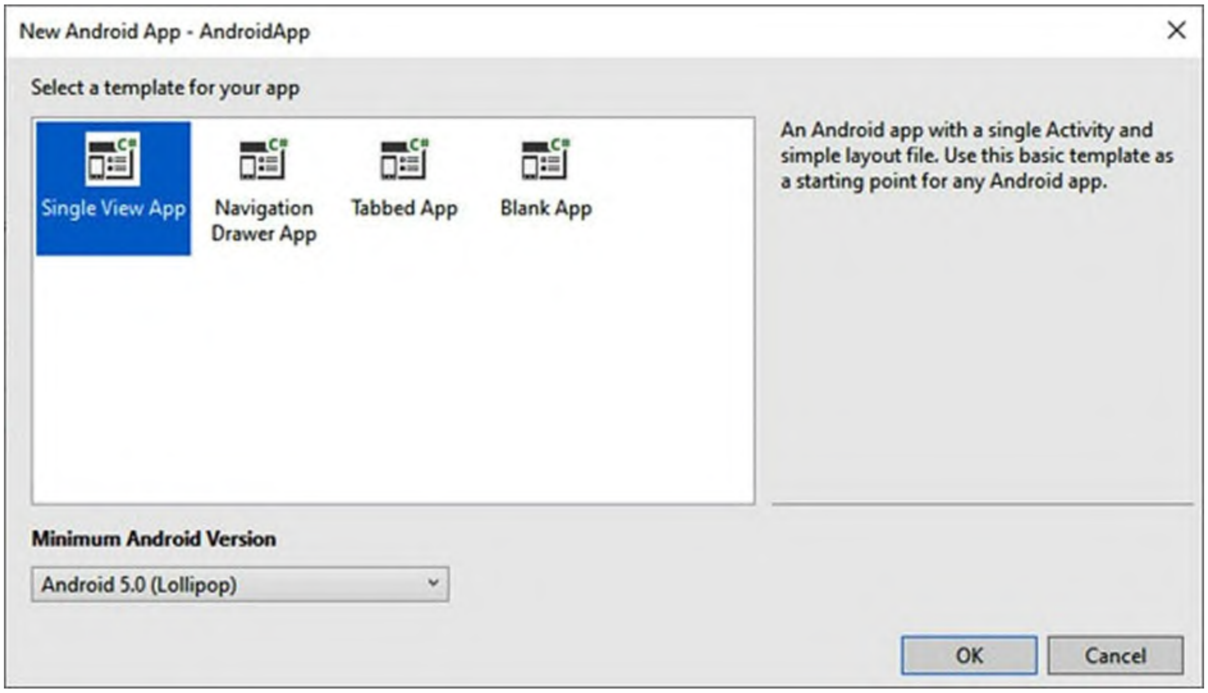
Android Bindings Library (Xamarin)
A project for creating a Xamarin.Android class library that binds to a native Java library.
C# Android Mobile

Class library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

Mobile App (Xamarin.Forms)

***Figure 5.1:*** *Creating a Xamarin.Android project*

Select the template and click on `Next`. At this point, you will be asked to specify an application type in the `New Android App` dialog, as shown in *Figure 5.2*:


New Android App - AndroidApp

Select a template for your app

An Android app with a single Activity and simple layout file. Use this basic template as a starting point for any Android app.

Minimum Android Version

Android 5.0 (Lollipop)

***Figure 5.2:*** *Selecting an Android app type*

*Table 5.1* describes the possible app types.

| Android App Type | Description |
|---|---|
| Single View App | Simple app based on one page. |
| Navigation Drawer App | App that implements navigation between multiple pages. |
| Tabbed App | App based on pages grouped by tabs. |
| Blank App | Empty projects for an app where you need to implement everything from scratch. |

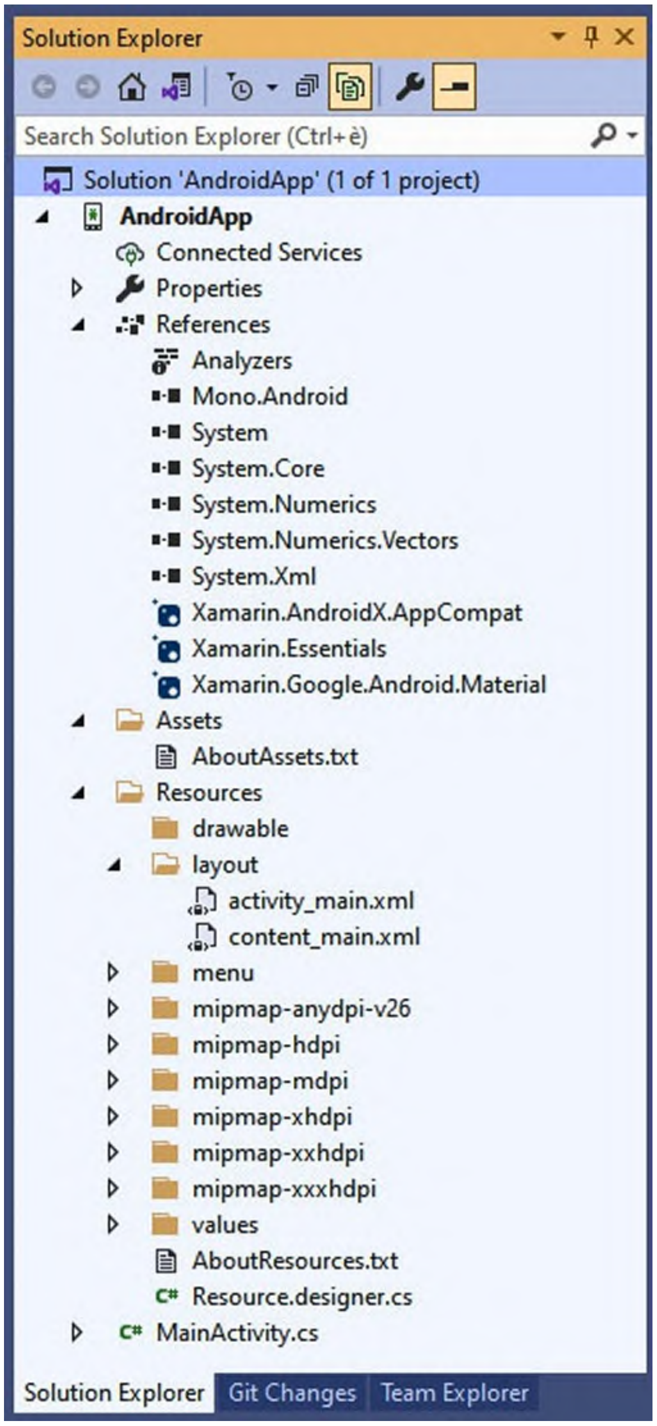
***Table 5.1:*** *Android app types in C#*

For demonstration purposes, select the **`Single View App`** template. Note how you can specify the minimum required Android version in the **`Minimum Android Version`** dropdown at the bottom left (see *Figure 5.2*). This is a crucial choice: selecting a higher version gives you the option to leverage the most recent and modern operating system API and device features but will exclude users who have a device with a lower version; selecting a lower version gives you the option to target a larger user base, but it requires you to use OS and device features that are available on the version you select, which might not be so recent. When you click on **`OK`**, you will be asked to specify a project name. Enter **`AndroidApp`** for this example. After a few seconds, the

Xamarin.Android solution will be ready.

## Understanding the project structure

In Solution Explorer, you can see the project structure, which looks like in *Figure 5.3*:



```
Solution 'AndroidApp' (1 of 1 project)
  AndroidApp
    Connected Services
    Properties
    References
      Analyzers
      Mono.Android
      System
      System.Core
      System.Numerics
      System.Numerics.Vectors
      System.Xml
      Xamarin.AndroidX.AppCompat
      Xamarin.Essentials
      Xamarin.Google.Android.Material
    Assets
      AboutAssets.txt
    Resources
      drawable
      layout
        activity_main.xml
        content_main.xml
      menu
      mipmap-anydpi-v26
      mipmap-hdpi
      mipmap-mdpi
      mipmap-xhdpi
      mipmap-xxhdpi
      mipmap-xxxhdpi
      values
      AboutResources.txt
      Resource.designer.cs
    MainActivity.cs
```

***Figure 5.3:*** *Structure of the Xamarin.Android project*

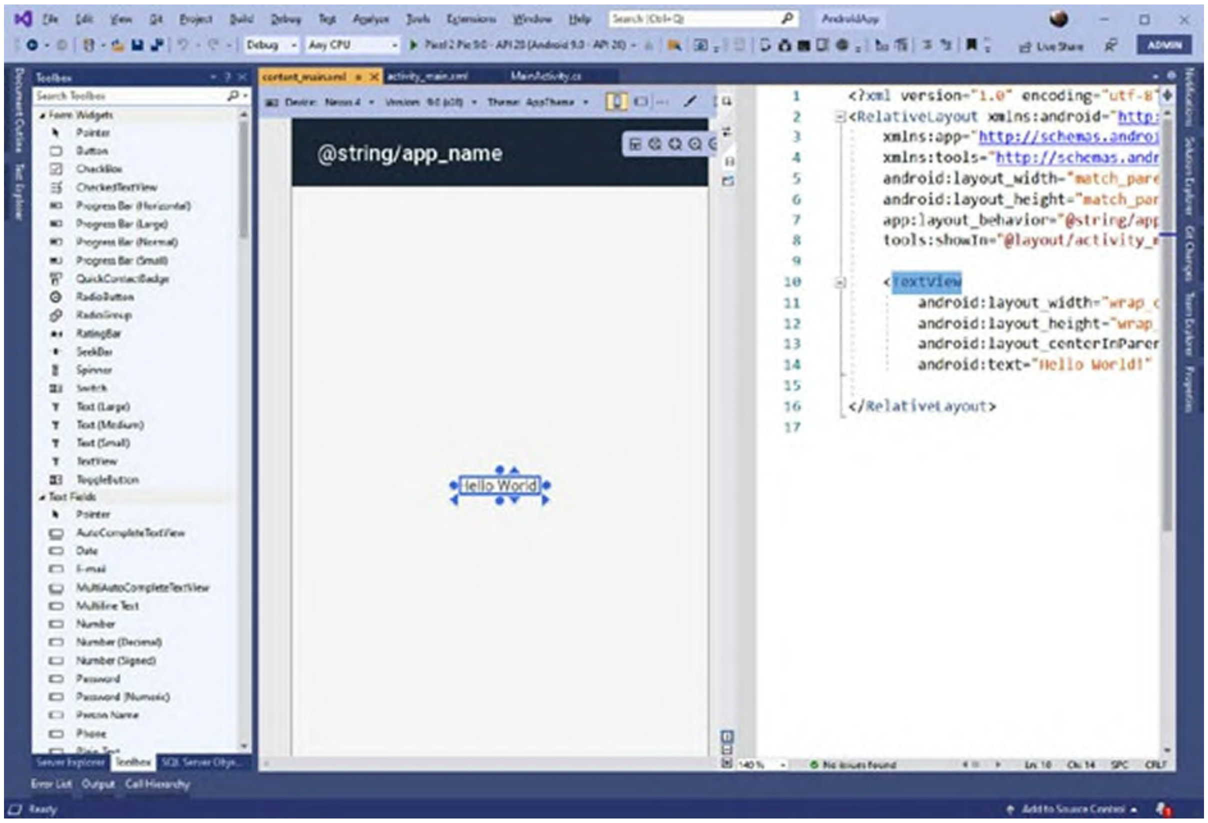
The following is a list of key points to remember:

- Under **`References`**, you will find the list of the required libraries, both system and third parties.
- Under the **`Assets`** folder, you can add resource files that the app needs, such as documents or videos.
- Under the **`Resources`** folder, you can find subfolders where you can add images, icons, and layout files. More specifically, you will add images into drawable and folders whose name starts with **`mipmap`**. There is a folder for each resolution.
- In the **`Resources\layout`** folder, you can find two XML files, which are related to the user interface definition.

The **`activity_main.xml`** file represents the main entry point of the app and acts as a container of pages, where you can also add controls that are common to multiple pages. In Android development, an activity represents either a single page or an individual functionality represented by a block of components. The **`content_main.xml`** file represents the main page, and this is the place where you can draw the user interface and handle its behavior.

## Drawing and handling the user interface

In Xamarin.Android, you draw the user interface by editing XML files, as you would do with native tools like Android Studio. If you double-click on the **`content_main.xml`** file, you will see the code editor and the graphic designer, where you can draw elements of the user interface. If you expand the ToolBox tool window (see *Figure 5.4*), you will also see a list of controls that can be used to draw the user interface on the designer:



```
<?xml version="1.0" encoding="utf-8"
<RelativeLayout xmlns:android="http:
    xmlns:app="http://schemas.androi
    xmlns:tools="http://schemas.andr
    android:layout_width="match_pare
    android:layout_height="match_par
    app:layout_behavior="@string/app
    tools:showIn="@layout/activity_

    <TextView
        android:layout_width="wrap_c
        android:layout_height="wrap_
        android:layout_centerInParer
        android:text="Hello World!"

</RelativeLayout>
```

***Figure 5.4:*** *Editing the user interface*

The ToolBox should be auto hidden on the left side of the IDE, but if you cannot see it there, you can select **`View`** | **`Toolbox`**. You can drag controls onto the designer surface. For example, drag a **`TextView`** control onto the designer. This control allows for displaying some text. When you do this, the code editor is updated with the corresponding XML markup. You can then edit text in the android:text property (see *Figure 4.5*) or in the **`Properties`** window.

**Describing the Android controls is not in the scope of this chapter; the goal here is to introduce you to Xamarin.Forms. However, as a general rule, remember that a control is represented by an XML element and its properties by XML attributes.**

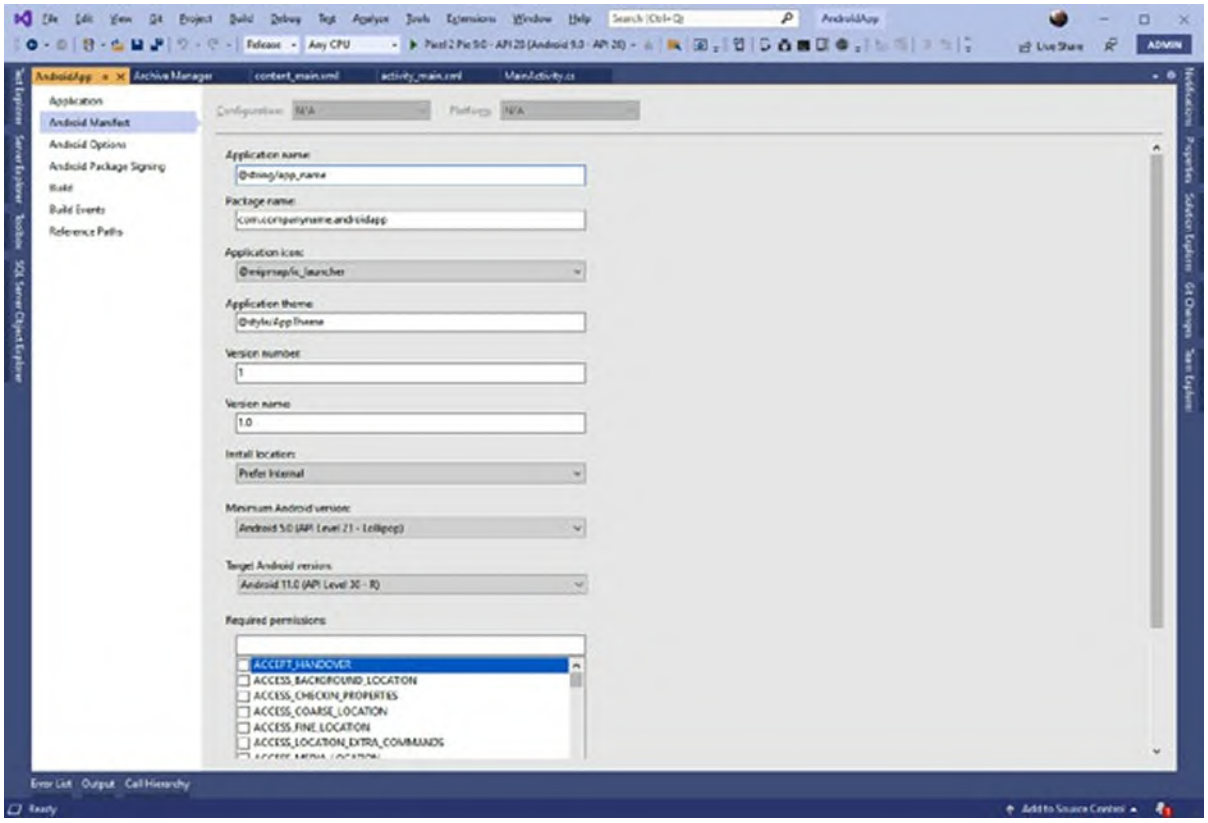
The user interface must react to the user actions. For example, the user taps a button or enters some text in a text box. Elements of the user interface raise events, which can be handled in C#. For example, if you open the **`main_activity.xml`** file, you will see that there is a floating button that allows for sending emails. In the **`MainActivity.cs`** file, a **`Click`** event for this button is handled as follows:

```
private void FabOnClick(object sender, EventArgs eventArgs)
{
  View view = (View) sender;
  Snackbar.Make(view, “Replace with your own action",
  Snackbar.LengthLong)
    .SetAction(“Action", (View.IOnClickListener)null).Show();
}
```

In short, when the user clicks on the button, the code retrieves the instance of the button itself and then invokes the **`Snackbar.Make`** method to take an action that must be implemented by the developer. For now, you should know that actions are handled in C#; in the upcoming chapters, you will see how this concept is widely applied to Xamarin.Forms, which is what this book focuses on. Then, handling events in native code is discussed in *Chapter 13: Working with Native APIs*.

## Configuring the App Manifest

An Android application can be configured in the so-called manifest. The manifest is an XML file that contains a set of properties that make it possible to configure the behavior of the app, and Visual Studio offers a convenient graphical editor for it. If you double-click on the **`Properties`** element in **`Solution Explorer`**, the **`Application Manifest`** will appear, as shown in *Figure 5.5*.



Configuration: N/A
Platform: N/A

Application name:
@string/app_name

Package name:
com.companyname.androidapp

Application icon:
@mipmap/ic_launcher

Application theme:
@style/AppTheme

Version number:
1

Version name:
1.0

Install location:
Prefer Internal

Minimum Android version:
Android 5.0 (API Level 21 - Lollipop)

Target Android version:
Android 11.0 (API Level 30 - R)

Required permissions:
ACCEPT_HANDOVER
ACCESS_BACKGROUND_LOCATION
ACCESS_CHECKIN_PROPERTIES
ACCESS_COARSE_LOCATION
ACCESS_FINE_LOCATION
ACCESS_LOCATION_EXTRA_COMMANDS

*Figure 5.5:* *Editing the application manifest*

Table 5.2 summarizes the options you can configure via the manifest:

| Property | Description |
|---|---|
| **`Application name`** | The name of the app as it appears on the device. It points to the `app_name` variable defined in the `Resources\values\strings.xml` file. |
| **`Package name`** | A unique identifier for the app. By convention it is com., followed by the company name and the app name. |
| **`Application icon`** | The application icon, a `.png` file stored under the `mipmap` folders. |
| **`Application theme`** | Optional custom graphical theme for the app. |
| **`Version number`** | An incremental, unique version number for the app. It can be 1, 2, 3, and so on. |
| **`Version name`** | The version number as it will appear to customers (for example, 1.0, 2.1, etc.). |
| **`Install location`** | With supported systems (up to Android 5.1), it allows for installing the app on external storages. |
| **`Minimum Android version`** | The minimum Android version required for the app to run. |

| | |
|---|---|
| **`Target Android version`** | The version of the Android SDK and tools used to compile the project. |
| **`Required permissions`** | This allows specifying the permissions the app needs to access system resources and device tools. |

***Table 5.2:*** *Manifest properties*

Take a moment to scroll through the **`Required permissions`** list. This is extremely important because an app is not free to access system resources and device tools like the camera or sensors, and this makes completely sense for both privacy reasons and awareness for the user about what the app can do. For example, if your app needs to connect to the internet, you will need to flag the **`INTERNET`** permission. If it allows for taking pictures or recording videos, the **`CAMERA`** permission must be selected. Permission names are very often self-explanatory, but you will learn more about permissions with the code examples provided in the upcoming chapters, where you will implement real application features. Permissions must be then handled in code to detect whether the user accepts or rejects permission requests. Luckily, the Xamarin.Android project template does this for you. In the **`MainActivity.cs`** file, you can find the following code:

```
public override void OnRequestPermissionsResult(int requestCode,
    string[] permissions, [GeneratedEnum]
    Android.Content.PM.Permission[]
    grantResults)
{
  Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestC
  permissions, grantResults);
  base.OnRequestPermissionsResult(requestCode, permissions,
  grantResults);
}
```

The **`OnRequestPermissionsResult`** method invokes the same-named method from the Platform class of the Xamarin.Essentials library, an important tool that provides API for common development tasks and which will be discussed in *Chapter 11, Managing the Application Lifecycle*. This method takes care of requesting the user permission based on what you have set in the manifest. If you forget to configure permissions in the manifest, the app will throw an exception, so you need to be very careful about this.

## Debugging an Android app

You can run, debug, and test Android apps using the steps you learned in the previous chapters, which means pressing *F5* to start debugging or *Ctrl* + *F5* to launch the app without a debugger attached. However, the target here is no longer the Windows' console, but a device. This can be a physical phone or tablet or an emulator. For the sake of simplicity, the following paragraphs are based on an emulator. In the Visual Studio toolbar, near the **`Start`** icon, you should see the name of a pre-configured Android emulator, and you should be able to click on the arrow down to see a list of available target devices (see *Figure 5.6*):

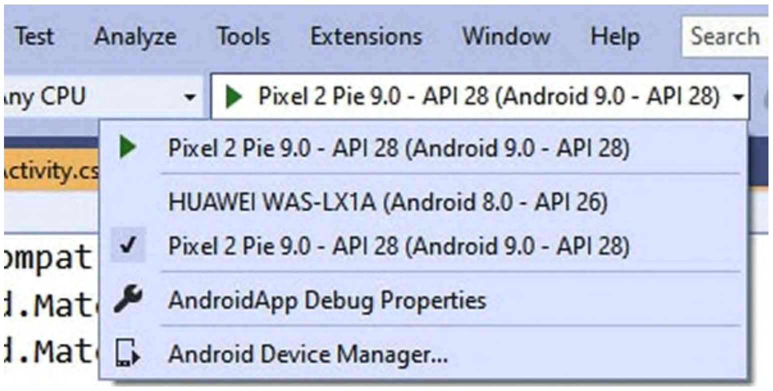
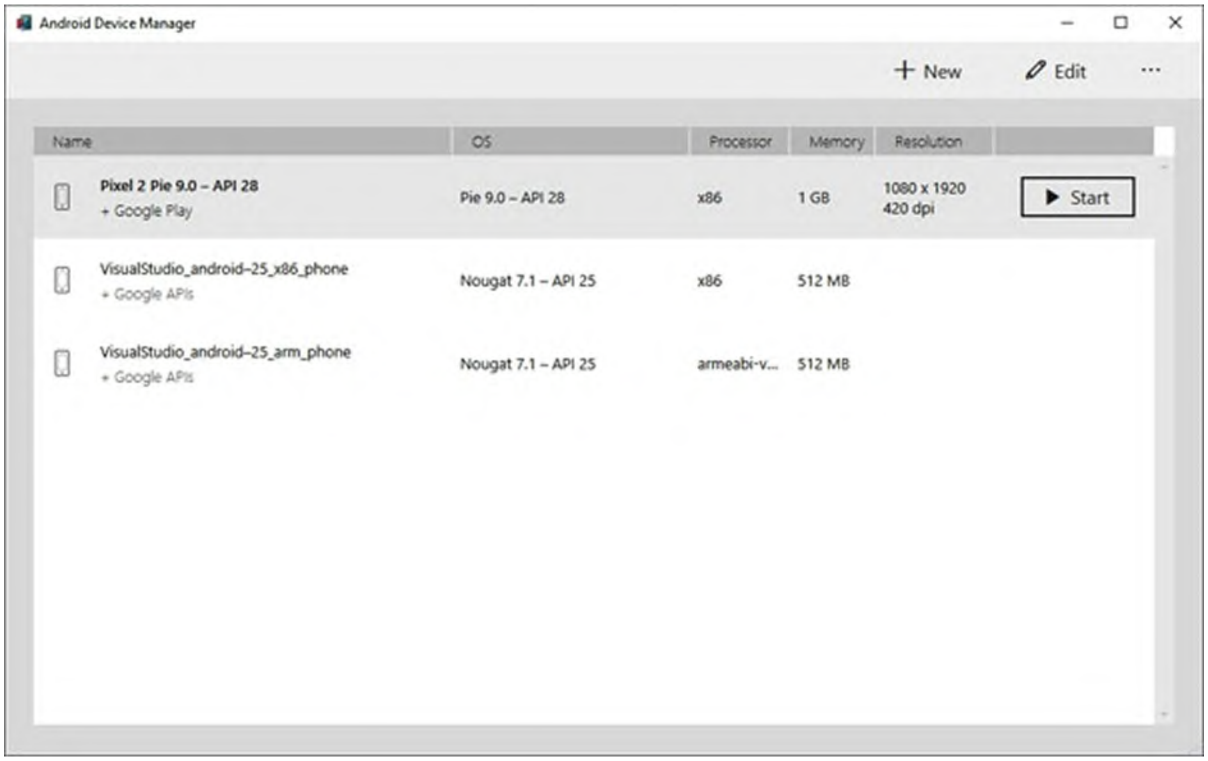

Test
Analyze
Tools
Extensions
Window
Help
Search
ny CPU
Pixel 2 Pie 9.0 - API 28 (Android 9.0 - API 28)
Pixel 2 Pie 9.0 - API 28 (Android 9.0 - API 28)
HUAWEI WAS-LX1A (Android 8.0 - API 26)
Pixel 2 Pie 9.0 - API 28 (Android 9.0 - API 28)
AndroidApp Debug Properties
Android Device Manager...

***Figure 5.6:*** *Selecting a device for debugging*

On my machine, the list shows an emulator (Pixel 2 Pie 9.0) and a physical device (**`HUAWEI WAS-LX1A`**). If you want to configure the existing emulators or create a new emulator image with your favorite system properties, you can click on **`Android Device Manager`**. This will open the Android Device Manager tool, as illustrated in *Figure 5.7*:



| Name | OS | Processor | Memory | Resolution | |
|---|---|---|---|---|---|
| Pixel 2 Pie 9.0 – API 28<br>+ Google Play | Pie 9.0 – API 28 | x86 | 1 GB | 1080 x 1920<br>420 dpi | ▶ Start |
| VisualStudio_android-25_x86_phone<br>+ Google APIs | Nougat 7.1 – API 25 | x86 | 512 MB | | |
| VisualStudio_android-25_arm_phone<br>+ Google APIs | Nougat 7.1 – API 25 | armeabi-v... | 512 MB | | |

***Figure 5.7:*** *The Android Device Manager tool*

For example, you can see that the Pixel 2 Pie emulator is configured with 1 GB of RAM memory and a screen resolution of 1080 x 1920. If you wish to change the configuration, you can click on **`Edit`**. Instead, if you prefer to create a new emulator image from scratch, you can click on **`New`**. This is out of scope here, so it is left as an exercise for you. You do not need to manually start the emulator from the Android Device Manager because Visual Studio does it for you. So, if you go back to Visual Studio and press the **`Start`** button, the selected emulator will start up, and the app will be loaded. If you are targeting a physical device, Visual Studio will launch the app on the device with an instance of the debugger attached. *Figure 5.8* shows how the current, simple app looks like on the emulator:

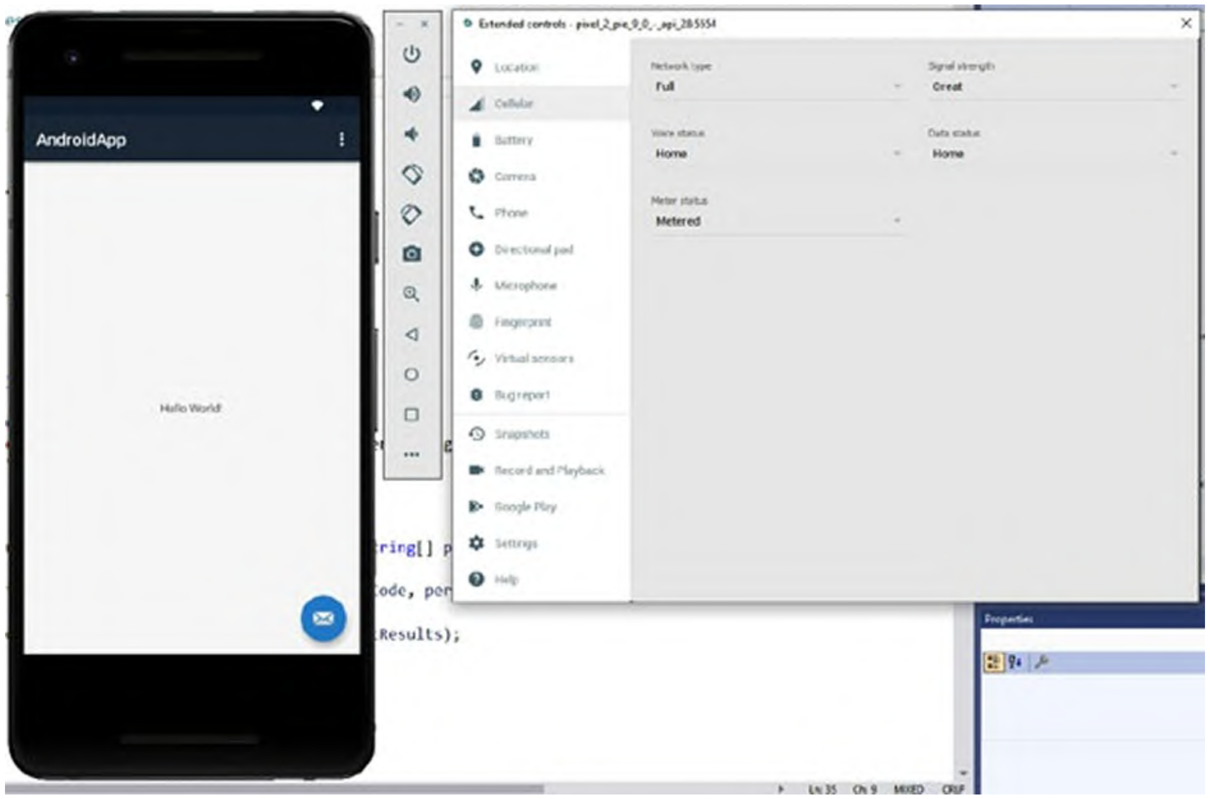


***Figure 5.8:*** *An Android app running in the emulator with extended controls*

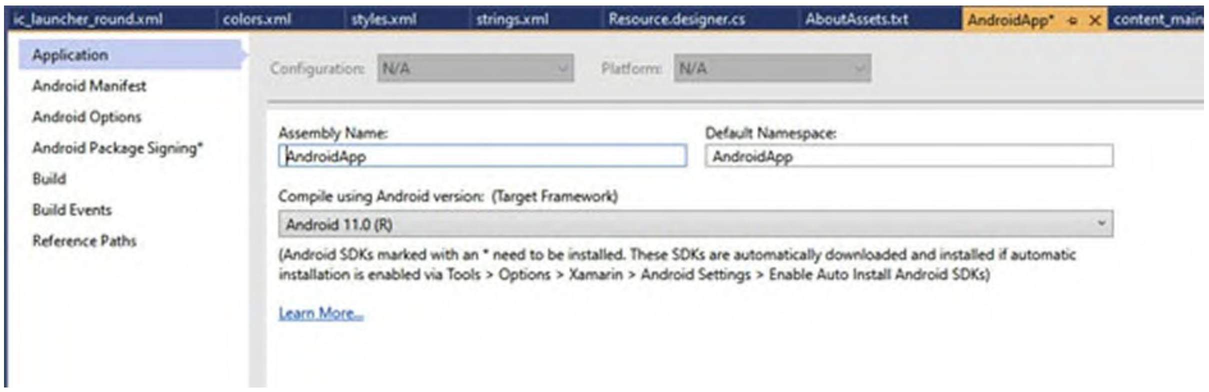
Emulators provide extended controls that you can use to simulate different situations in the application lifecycle, such as the battery level, network connection, and location. You enable extended controls by clicking on the … button at the bottom of the emulator toolbar. You can also rotate the emulator, control the audio level, and take screenshots. When debugging, you can use all the tools you learned in *Chapter 3, Introducing Visual Studio and .NET*, such as breakpoints and data tips.

**When the debugger is attached to the app, the execution is certainly much slower. The reason is that the debugger needs to walk through every single line of code that is executed, including system code run in the background. If you wish to test the real app performance, you should run the app at least without the debugger, and possibly, on a physical device.**

The resulting **`.apk`** files can be found under **`bin\debug`** or **`bin\release`**, depending on the selected build configuration. You will find two **`.apk`** files, and one contains the signed literal in its name. If you do not provide a signature, Visual Studio adds one for debugging purposes on a physical device. Regardless of the device you use, you now have the option to see how your Android app works, and this is what you will exactly do with Xamarin.Forms across the rest of the book and with a real user interface, so keep these steps in mind.

## Configuring debugging and Build options

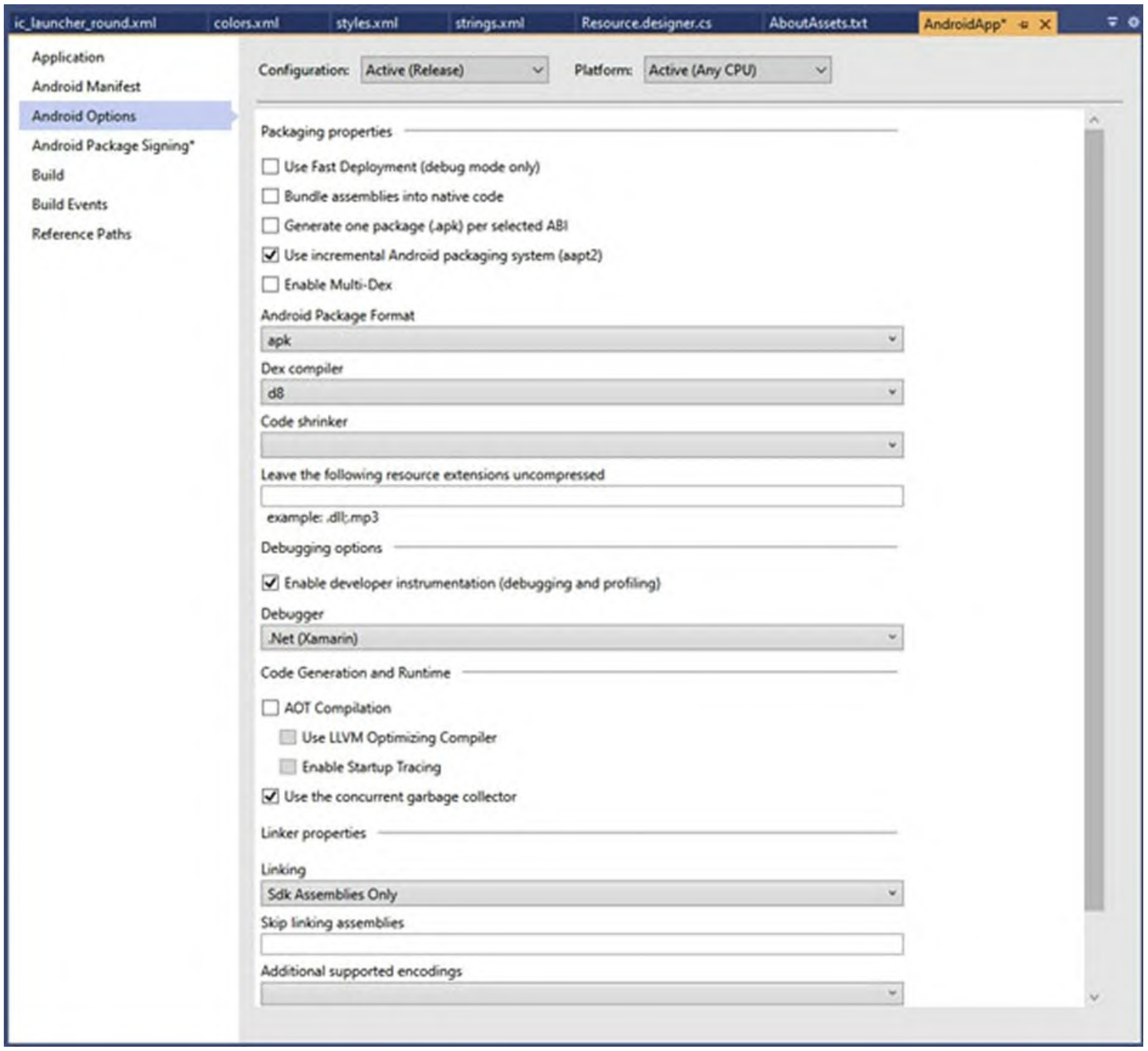
Visual Studio gives you the option to configure several options for debugging and building an Android app package. If you open the project properties, you will be able to edit the version of the Android SDK used to build the app package in the `Application` tab, as shown in *Figure 5.9*:



Configuration: N/A
Platform: N/A
Assembly Name: AndroidApp
Default Namespace: AndroidApp
Compile using Android version: (Target Framework) Android 11.0 (R)
(Android SDKs marked with an * need to be installed. These SDKs are automatically downloaded and installed if automatic installation is enabled via Tools > Options > Xamarin > Android Settings > Enable Auto Install Android SDKs)

***Figure 5.9:*** *Configuring the Android SDK version*

This must not be confused with the minimum version required to run your app. Instead, this allows you to specify which version of the Android tools must be used to build the app package. When possible, you will use the latest version available on your machine, but you can downgrade in case of compatibility issues. In the `Android Options` tab, you can configure debugging and build options. If you look at *Figure 5.10*, you will see many options, but only a subset will be relevant for your daily work:



Configuration: Active (Release) Platform: Active (Any CPU)

Packaging properties

- [ ] Use Fast Deployment (debug mode only)
- [ ] Bundle assemblies into native code
- [ ] Generate one package (.apk) per selected ABI
- [x] Use incremental Android packaging system (aapt2)
- [ ] Enable Multi-Dex

Android Package Format: apk

Dex compiler: d8

Code shrinker:

Leave the following resource extensions uncompressed:

example: .dll;.mp3

Debugging options

- [x] Enable developer instrumentation (debugging and profiling)

Debugger: .Net (Xamarin)

Code Generation and Runtime

- [ ] AOT Compilation
  - [ ] Use LLVM Optimizing Compiler
  - [ ] Enable Startup Tracing
- [x] Use the concurrent garbage collector

Linker properties

Linking: Sdk Assemblies Only

Skip linking assemblies:

Additional supported encodings:

***Figure 5.10:*** *Configuring debugging and build options*

*Table 5.3* summarizes the options that are relevant for you:

| Option | Description |
|---|---|
| **`Use Fast Deployment (debug mode only)`** | When enabled, only the updated assemblies are rebuilt. This also keeps the data in the app cache. |
| **`Android Package Format`** | You can decide between the classic `.apk` format and the new app bundle, which creates a smaller package and delegates the `.apk` creation to the Google Play. |
| **`Linking`** | This allows for controlling the linker, a tool that continuously looks for unused types and references and removes them to keep the package size smaller. Supported options are Sdk Assemblies only (default), Sdk and User Assemblies, and None. |

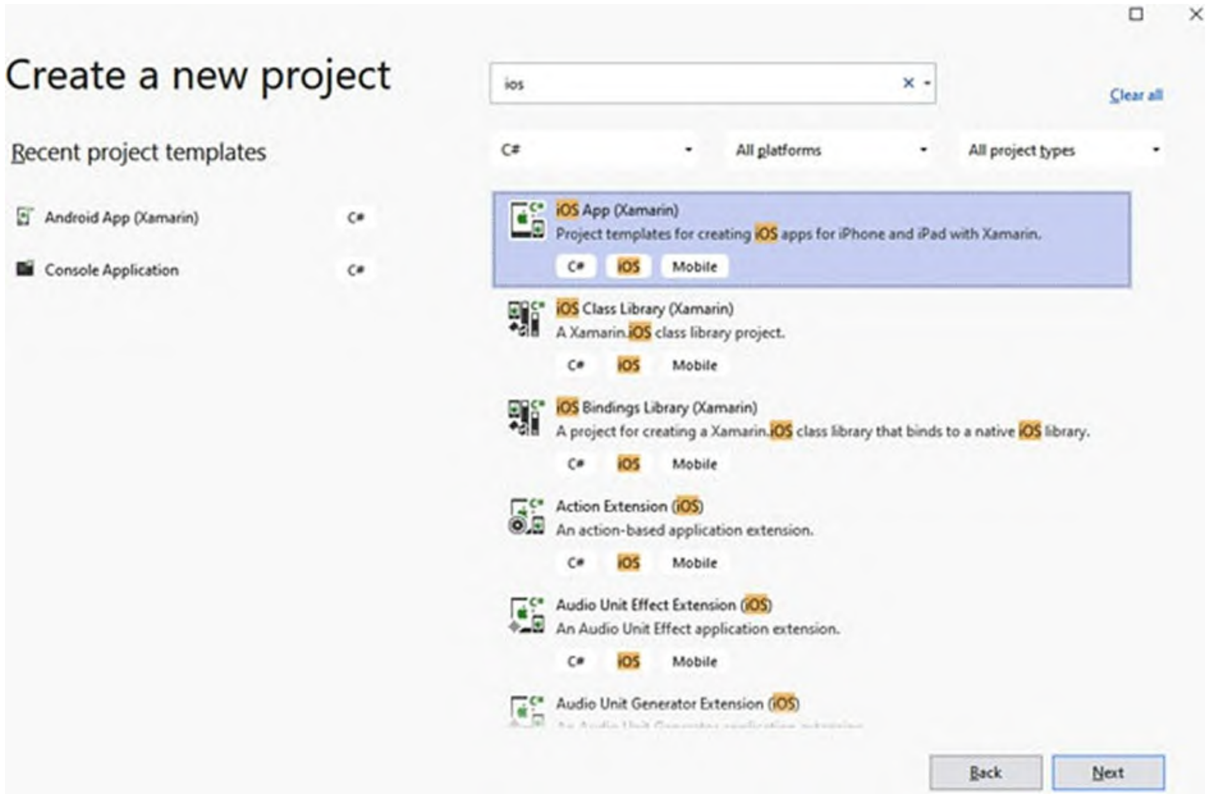
*Table 5.3: Most relevant debugging and build options*

For now, leave the default options. You can change them when required. All these concepts apply to the native Xamarin.Android project included in every Xamarin.Forms solution, so it is important that you know about them.

**The `Android Package Signing` options will be discussed in the *Publishing apps* section.**

## Understanding Xamarin.iOS projects

With Visual Studio, you can also create native Xamarin.iOS projects. Additionally, Xamarin.Forms solutions include a Xamarin.iOS native project if you decide to target this platform as well, so it is important for you to know at least about their structure. In the **`Create new project`** dialog, locate the **`iOS App (Xamarin)`** project template (see *Figure 5.11*):


Create a new project

Recent project templates

Android App (Xamarin) C#

Console Application C#

ios

C# · All platforms · All project types

iOS App (Xamarin)
Project templates for creating iOS apps for iPhone and iPad with Xamarin.
C# iOS Mobile

iOS Class Library (Xamarin)
A Xamarin.iOS class library project.
C# iOS Mobile

iOS Bindings Library (Xamarin)
A project for creating a Xamarin.iOS class library that binds to a native iOS library.
C# iOS Mobile

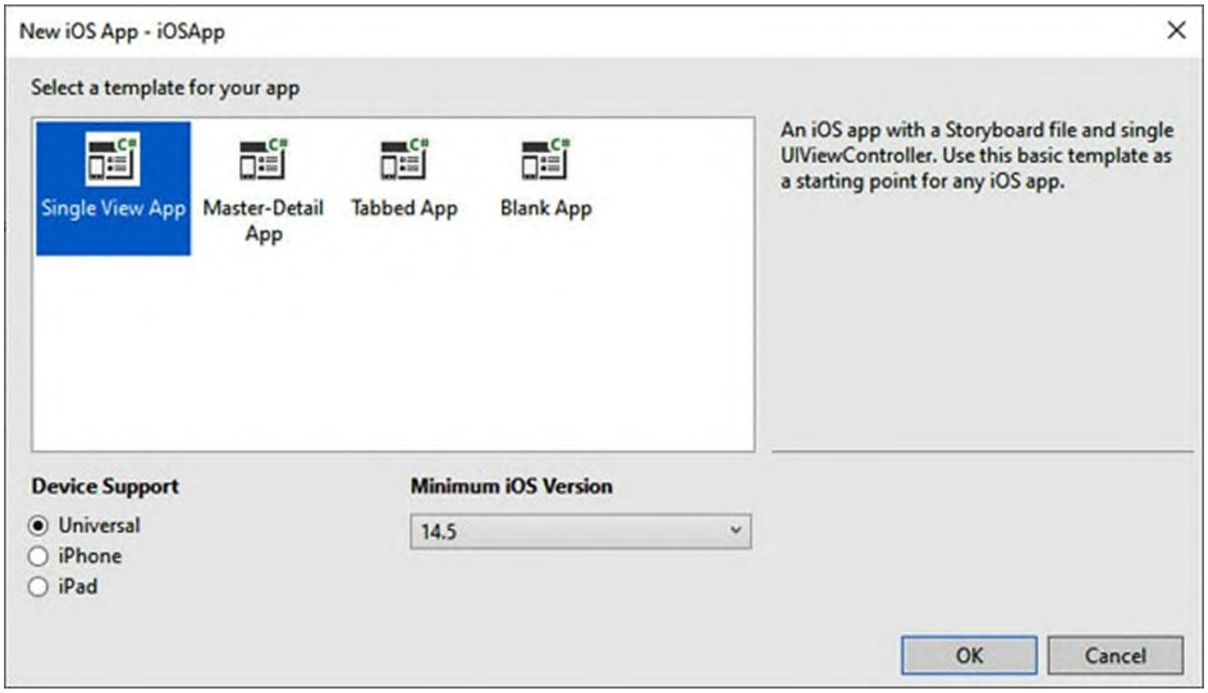
Action Extension (iOS)
An action-based application extension.
C# iOS Mobile

Audio Unit Effect Extension (iOS)
An Audio Unit Effect application extension.
C# iOS Mobile

Audio Unit Generator Extension (iOS)

***Figure 5.11:*** *Creating a Xamarin.iOS project*

You can use the search box to quickly filter the list. When you click on **`Next`**, you will be asked to specify a project type (see *Figure 5.12*).


New iOS App - iOSApp

Select a template for your app

An iOS app with a Storyboard file and single UIViewController. Use this basic template as a starting point for any iOS app.

Device Support

[✓ Universal] iPhone iPad

Minimum iOS Version: 14.5

***Figure 5.12:*** *Specifying the iOS project type*

*Table 5.4* describes the possible app types:

| Android App Type | Description |
|---|---|
| `Single View App` | Simple app based on one page. |
| `Master-Detail App` | App that implements a main page (Master) with navigation between child pages (Detail). |
| `Tabbed App` | App based on pages grouped by tabs. |
| `Blank App` | Empty projects for an app where you need to implement everything from scratch. |

***Table 5.4:*** *iOS app types in C#*

Depending on which type of device you intend to target, you can select one in the Device Support group. The default is Universal, which makes an app work on both iPhones and iPads. The type of device can be always changed in the **`Info.plist`** file in a second moment. The **`Minimum iOS Version`** dropdown allows you to select which minimum iOS version must be installed on the target device. The lower the version, the higher the number of users you can reach, but you will not be able to leverage the API exposed by most recent versions. For the current example, select the **`Single View App`** project and then click on **`OK`**. In the next dialog, you will be able to specify a project name. For the current example, simply enter **`iOSApp`** and then click on **`OK`**. After a few seconds, the project will be ready, and Visual Studio will ask you to connect to a Mac computer. As you remember from *Chapter 1, The Importance of Mobile App Development*, and *Chapter 2, Xamarin and Microsoft in the Mobile App Market*, a Mac computer with the XCode development tools is required to build iOS apps with Xamarin and C#. In the **`Pair to Mac`** window, you can specify a Mac that must be in the same network or be accessible via its IP address. *Figure 5.13* demonstrates how this dialog appears:

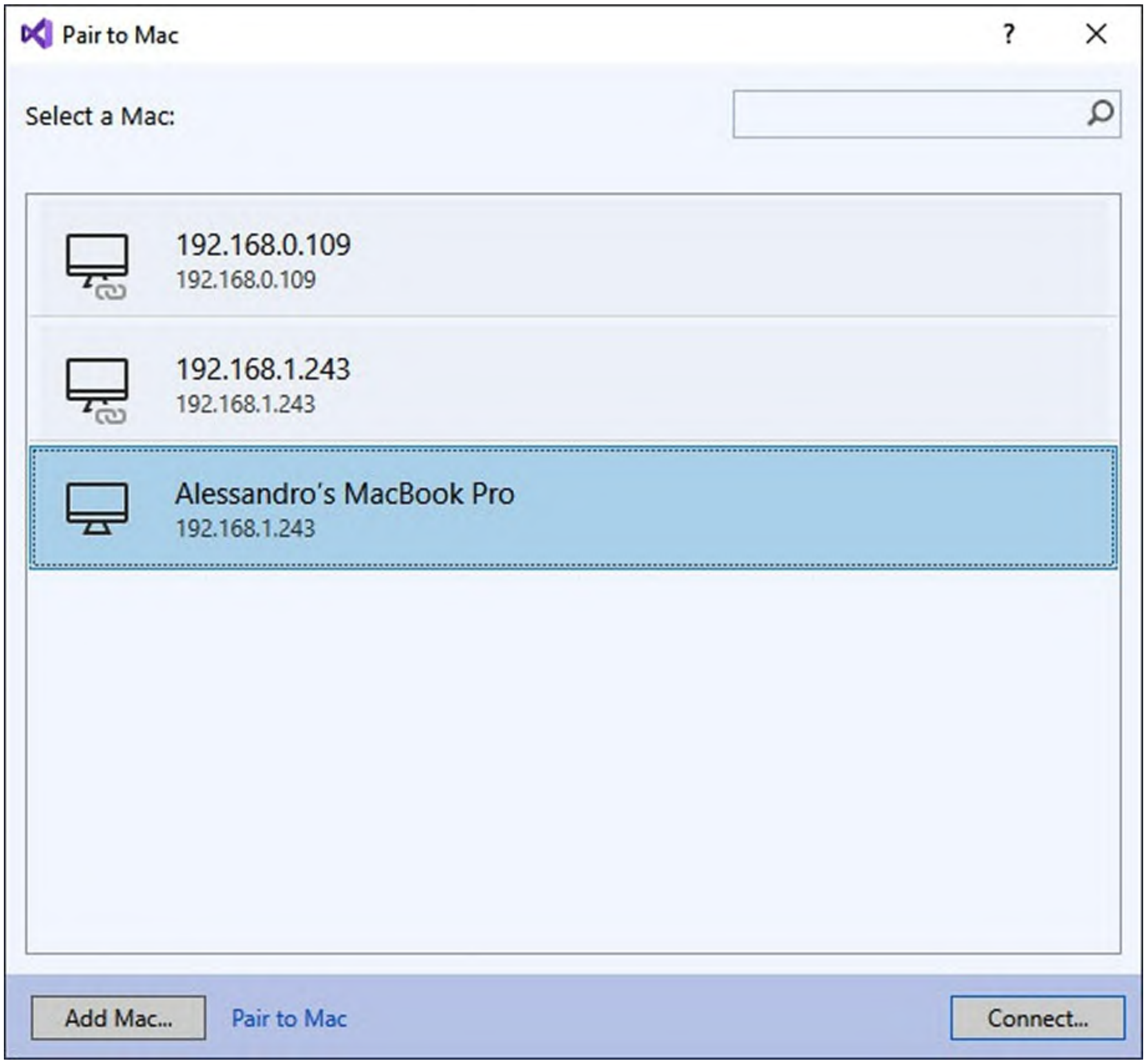


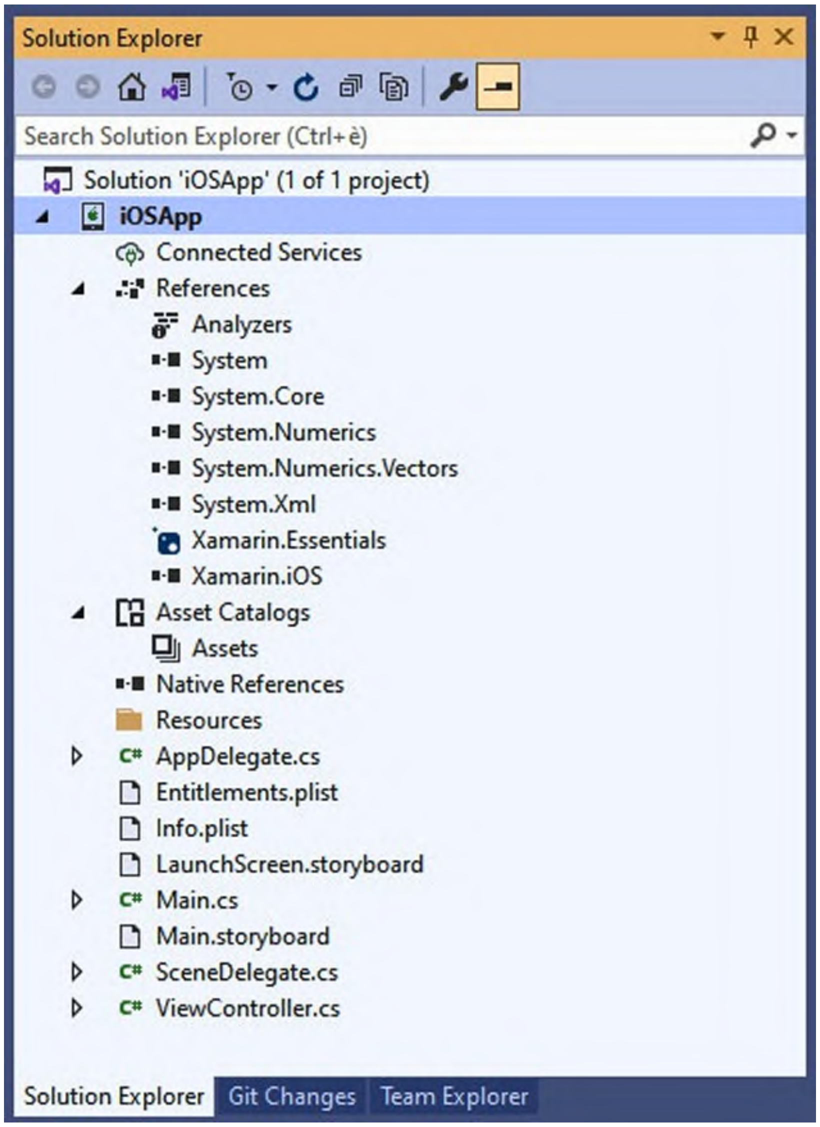
***Figure 5.13:*** *Connecting Visual Studio to a Mac computer*

**If the Pair to Mac dialog does not appear automatically, you can display it with Tools, iOS, Par to Mac.**

If your Mac appears in the list, just double-click on it or click on **`Add Mac`** to enter the name or IP address of the Mac itself. When ready, click on **`Connect`**. Assuming that you have enabled the remote access options on your Mac, Visual Studio will ask the Mac credentials to connect. These are the username and password of a Mac user that can login to the system. The dialog will finally inform you whether the connection succeeds or fails.

## Understanding the project structure

In Solution Explorer, you can see the project structure, which looks like in *Figure 5.14*:



```
Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'iOSApp' (1 of 1 project)
◢ iOSApp
    Connected Services
  ◢ References
      Analyzers
      System
      System.Core
      System.Numerics
      System.Numerics.Vectors
      System.Xml
      Xamarin.Essentials
      Xamarin.iOS
  ◢ Asset Catalogs
      Assets
    Native References
    Resources
  ▷ AppDelegate.cs
    Entitlements.plist
    Info.plist
    LaunchScreen.storyboard
  ▷ Main.cs
    Main.storyboard
  ▷ SceneDelegate.cs
  ▷ ViewController.cs
Solution Explorer   Git Changes   Team Explorer
```

***Figure 5.14:*** *The structure of a Xamarin.iOS project*

The following is a list of key points to remember:

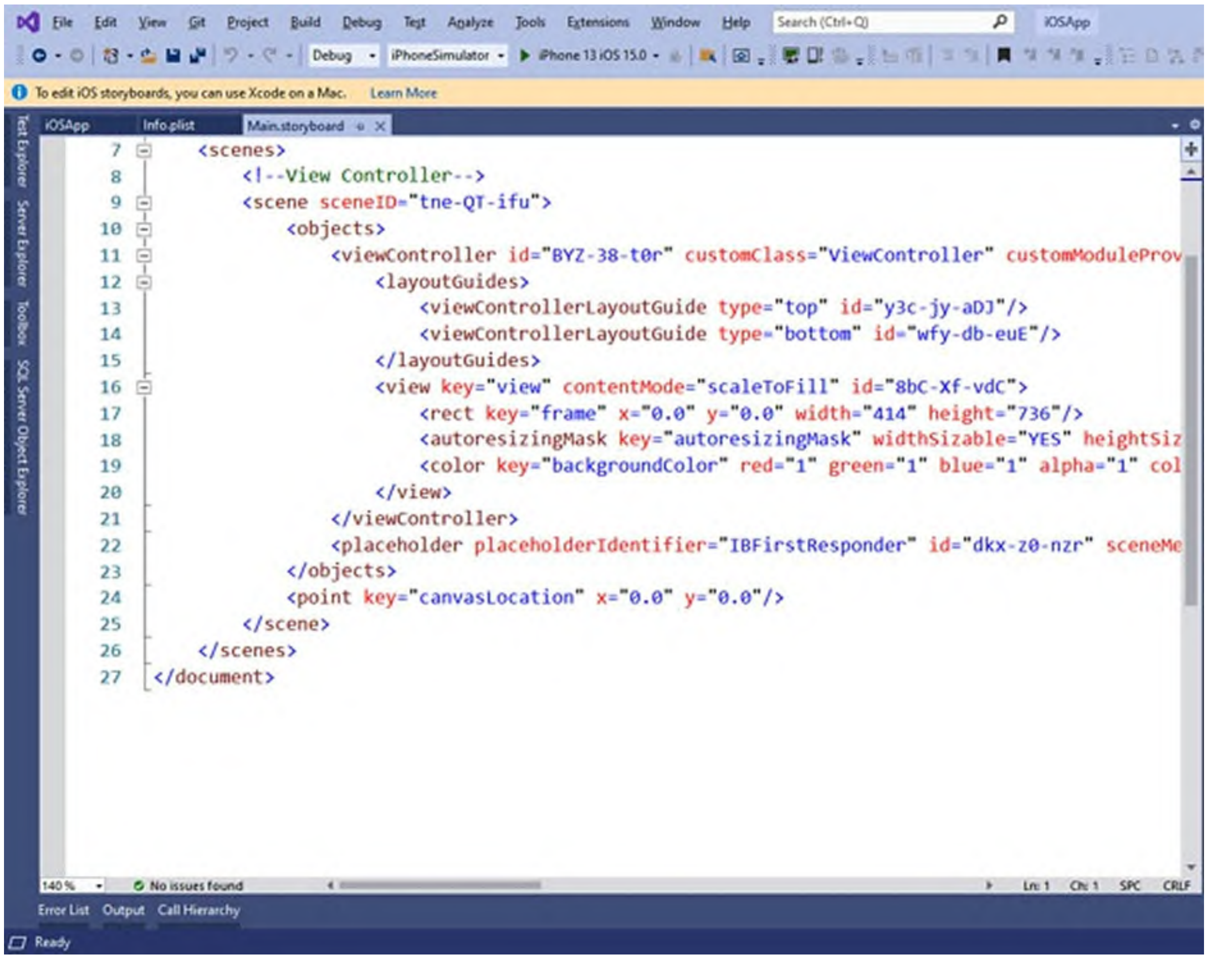
- Under **`References`**, you will find the list of the required libraries, both system and third parties.
- Under the **`Assets Catalogs`** folder, you store all the icons required by the Apple Store.
- Under the **`Resources`** folder, you add images, font files, documents, and any other resources that the app might need.
- The **`Info.plist`** file can be considered as the app manifest in iOS and allows for configuring the app's main properties. It is discussed in further detail in the upcoming sections.
- The **`Entitlements.plist`** file makes it possible to connect the app to several Apple services (for example, Apple Pay, Apple Sign-in, Siri, iCloud, etc.) and set up the proper authorizations.

You can also see two **`.storyboard`** files, which represent where the user interface is designed.

## Handling the user interface

Xamarin.iOS allows you to create the user interface with **`.storyboard`** files, which is exactly the same format utilized by Apple Xcode. If you look at *Figure 5.14*, you can see two files: **`LaunchScreen.storyboard`** and **`Main.storyboard`**. The first file allows you to create a functional splash screen for your app, which includes static images and also other kinds of visual elements. **`Main.storyboard`** represents the root page of the app, and it is the first page you create. Previously, Visual Studio had an integrated visual designer to help you create the user interface for Xamarin.iOS projects, but it has been recently removed. So, you have two options:

- Manually writing the XML markup that map the visual elements directly inside the **`.storyboard`** files. For a better understanding, you can double-click one of the **`.storyboard`** files and see how they appear in the code editor (see *Figure 5.15* for an example based on **`Main.storyboard`**).
- Editing **`.storyboard`** files in XCode on a Mac to leverage more sophisticated visual design tools.



```
    <scenes>
        <!--View Controller-->
        <scene sceneID="tne-QT-ifu">
            <objects>
                <viewController id="BYZ-38-t0r" customClass="ViewController" customModuleProv
                    <layoutGuides>
                        <viewControllerLayoutGuide type="top" id="y3c-jy-aDJ"/>
                        <viewControllerLayoutGuide type="bottom" id="wfy-db-euE"/>
                    </layoutGuides>
                    <view key="view" contentMode="scaleToFill" id="8bC-Xf-vdC">
                        <rect key="frame" x="0.0" y="0.0" width="414" height="736"/>
                        <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSiz
                        <color key="backgroundColor" red="1" green="1" blue="1" alpha="1" col
                    </view>
                </viewController>
                <placeholder placeholderIdentifier="IBFirstResponder" id="dkx-z0-nzr" sceneMe
            </objects>
            <point key="canvasLocation" x="0.0" y="0.0"/>
        </scene>
    </scenes>
</document>
```

***Figure 5.15:*** *The XML markup of a .storyboard file*

For the purpose of this book, both options are quite irrelevant because you will focus on Xamarin.Forms projects, and Visual Studio has proper design tools for this. But if you ever need to work on Xamarin.iOS projects with visual design tools, these will need to be opened on a Mac with XCode. Like for Android, Xamarin.iOS relies on C# code for responding to user actions on the user interface as well as for code that is required by the app lifecycle. If you open the various .cs files in the solution, you will see method names that easily recall events in the app or page lifecycle. You will learn more on this in *Chapter 13: Working with Native APIs*. For now, it is better to spend a few words on a real-world tip. Open the **`Main.cs`** file. This is the main entry point of the app and the first piece of code that will be executed. You will find a **`Main`** method defined as follows:

```
 static void Main(string[] args)
 {
   // if you want to use a different Application Delegate class
```

```
  from
  “AppDelegate"
  // you can specify it here.
  UIApplication.Main(args, null, “AppDelegate");
 }
```

The **UIApplication.Main** method creates an instance of the **UIApplication** class, which represents the application during all its lifecycle. With Xamarin.iOS, several types of runtime errors might make the app crash without any exceptions or error messages even with the Debug configuration enabled, making it extremely difficult to understand what caused the crash. For this reason, you can enclose the method invocation inside a **try..catch** block, as follows:

```
try
{
 UIApplication.Main(args, null, “AppDelegate");
}
catch (System.Exception ex)
{
 System.Diagnostics.Debug.WriteLine(ex.Message);
}
```

This way, if a problem happens, you can use a breakpoint on the **catch** block or look at the **Output** window to see the message provided by the **Exception** class. This also applies to your Xamarin.Forms projects, which rely on Xamarin.iOS for the iOS part, so this is a recommendation that you should implement on every native iOS project.

## Debugging an iOS app

In Xamarin.iOS, you have the usual **Debug** and **Release** build configurations, plus two additional flavors called **iPhoneSimulator** and iPhone. You use the first one when you want to work with one of the simulators provided by the XCode developer tools and the second one when you want to run the app on a physical device. For example, suppose you want to debug the current sample app on an iPhone 13 with iOS 15 installed. You select the **Debug configuration**, then **iPhoneSimulator** and the desired device from the dropdown placed on the Start button, as shown in *Figure 5.16*:

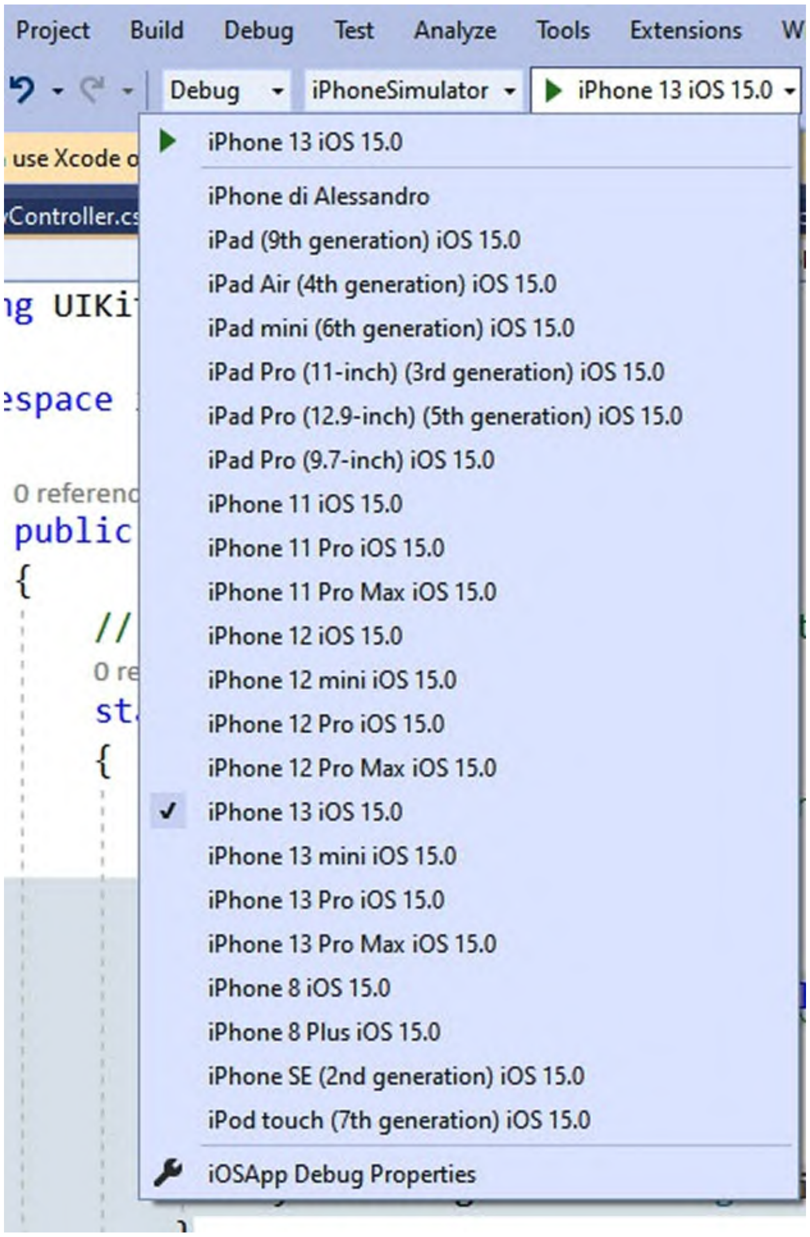

Project
Build
Debug
Test
Analyze
Tools
Extensions
Debug
iPhoneSimulator
iPhone 13 iOS 15.0
iPhone 13 iOS 15.0
iPhone di Alessandro
iPad (9th generation) iOS 15.0
iPad Air (4th generation) iOS 15.0
iPad mini (6th generation) iOS 15.0
iPad Pro (11-inch) (3rd generation) iOS 15.0
iPad Pro (12.9-inch) (5th generation) iOS 15.0
iPad Pro (9.7-inch) iOS 15.0
iPhone 11 iOS 15.0
iPhone 11 Pro iOS 15.0
iPhone 11 Pro Max iOS 15.0
iPhone 12 iOS 15.0
iPhone 12 mini iOS 15.0
iPhone 12 Pro iOS 15.0
iPhone 12 Pro Max iOS 15.0
✓ iPhone 13 iOS 15.0
iPhone 13 mini iOS 15.0
iPhone 13 Pro iOS 15.0
iPhone 13 Pro Max iOS 15.0
iPhone 8 iOS 15.0
iPhone 8 Plus iOS 15.0
iPhone SE (2nd generation) iOS 15.0
iPod touch (7th generation) iOS 15.0
iOSApp Debug Properties

***Figure 5.16:*** *Selecting a target device for debugging*

Physical devices connected to your Mac will also be displayed in this list, but you should pick them up only with the iPhone configuration flavor selected.

**You can install or remove device simulators from XCode on your Mac. This is not something that Visual Studio can handle.**

If you press *F5*, Visual Studio will contact the XCode developer tools on your Mac. This is required to first build the **`.ipa`** application package and then to start the iPhone simulator. Remember that everything happens on the

XCode side. After one minute or so, you should see the device simulator with the sample application running, as shown in *Figure 5.17*:

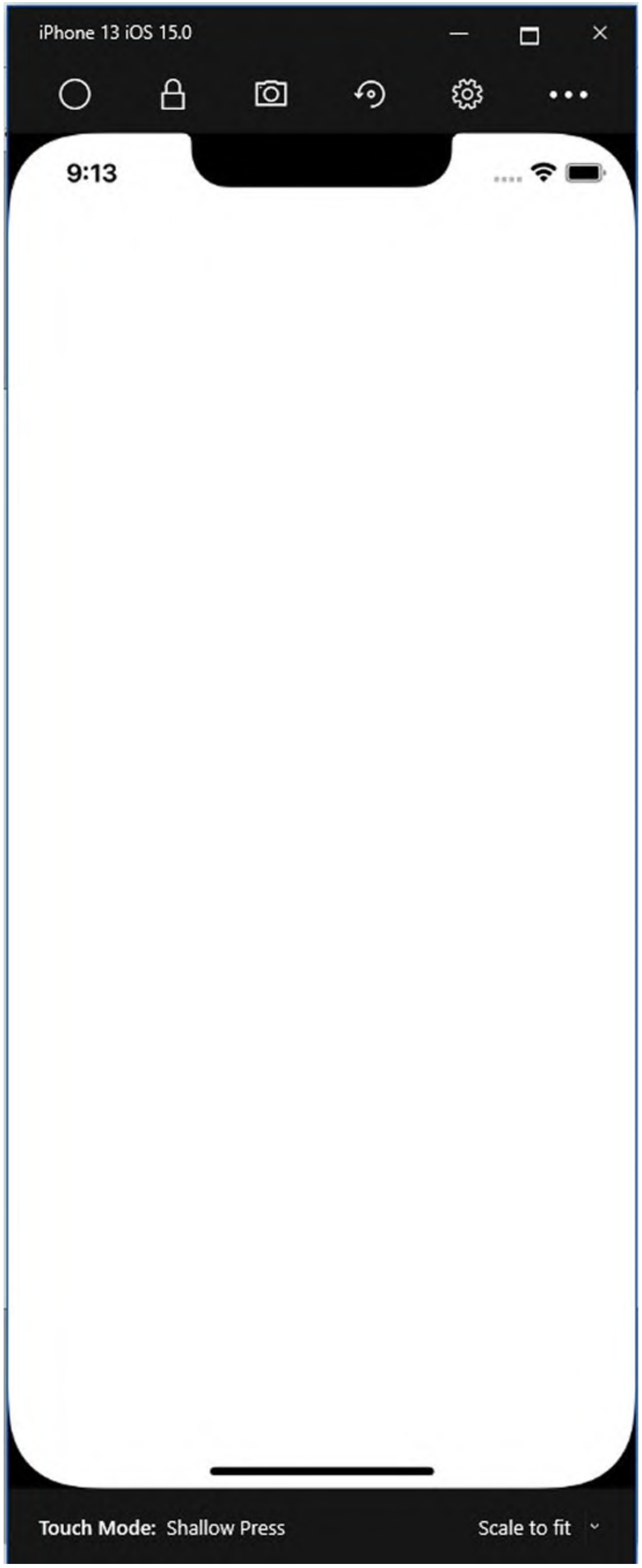

iPhone 13 iOS 15.0
9:13
Touch Mode: Shallow Press
Scale to fit

***Figure 5.17:*** *The empty Xamarin.iOS project is running on an iPhone 13 simulator*

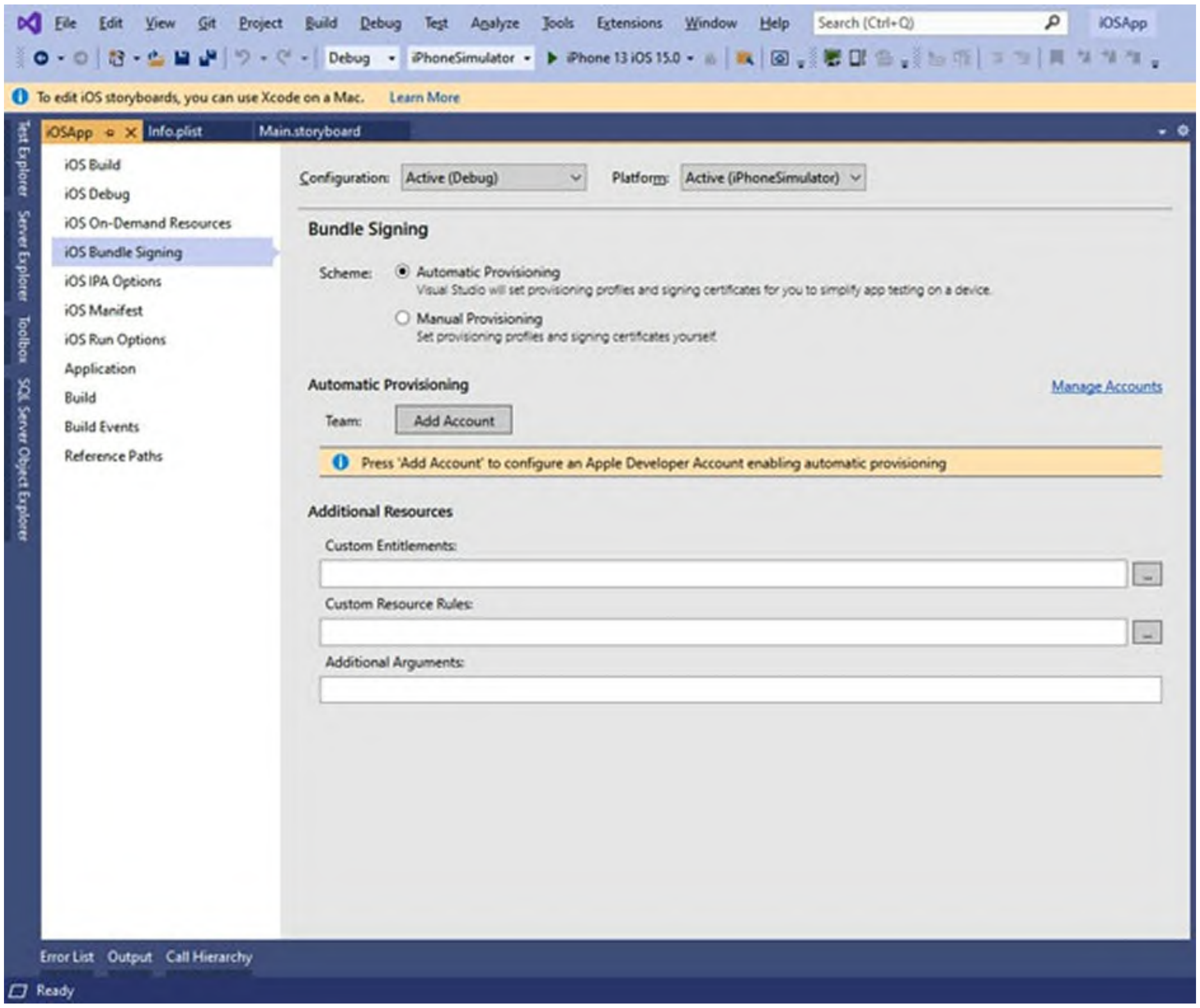
In practice, the simulator is running on the Mac. Visual Studio is only sharing the simulator on Windows with a dedicated user interface.

**Because iOS simulators actually run on a Mac, if you do not see also the simulator running on Windows, you must explicitly enable this option. This can be done by clicking on `Tools | Options | Xamarin | iOS Settings` and then selecting the option called `Remote Simulator to` Windows.**

The simulator has a toolbar with buttons that allow you to simulate gestures, such as pressing the physical home button, locking the screen, and taking screenshots. You can access advanced options by clicking on the … button, such as setting the network conditions and a sample location. Working with simulators not only makes it possible to test the app with different screen factors and hardware features, but it also simplifies the development process because it does not need to have a developer profile. When you want to debug an app on a physical device, you need a developer profile. This is explained in the next section.

## Understanding provisioning profiles and developer accounts

Building apps for iOS requires an Apple developer account. For development and debugging purposes, the Apple SDKs generates a free developer account based on your Apple ID that you can use to write code and debug. This happens automatically. As mentioned in the *Publishing apps* section later, you will need a paid developer account to publish apps or to distribute apps to testers. For now, the free individual account is fine. If you wish to run your apps on a physical device, you also need so-called **provisioning profiles**. A provisioning profile allows a Mac to recognize your physical devices for development purposes. This is accomplished in the project properties, so right-click on the project name in **`Solution Explorer`** and then select **`Properties`**. The **`iOS Bundle Signing`** tab should be automatically opened (see *Figure 5.18*):



To edit iOS storyboards, you can use Xcode on a Mac. Learn More

Configuration: Active (Debug) Platform: Active (iPhoneSimulator)

Bundle Signing

Scheme: [✓ Automatic Provisioning] Visual Studio will set provisioning profiles and signing certificates for you to simplify app testing on a device.
Manual Provisioning Set provisioning profiles and signing certificates yourself.

Automatic Provisioning

Team:

Press 'Add Account' to configure an Apple Developer Account enabling automatic provisioning

Additional Resources

Custom Entitlements:

Custom Resource Rules:

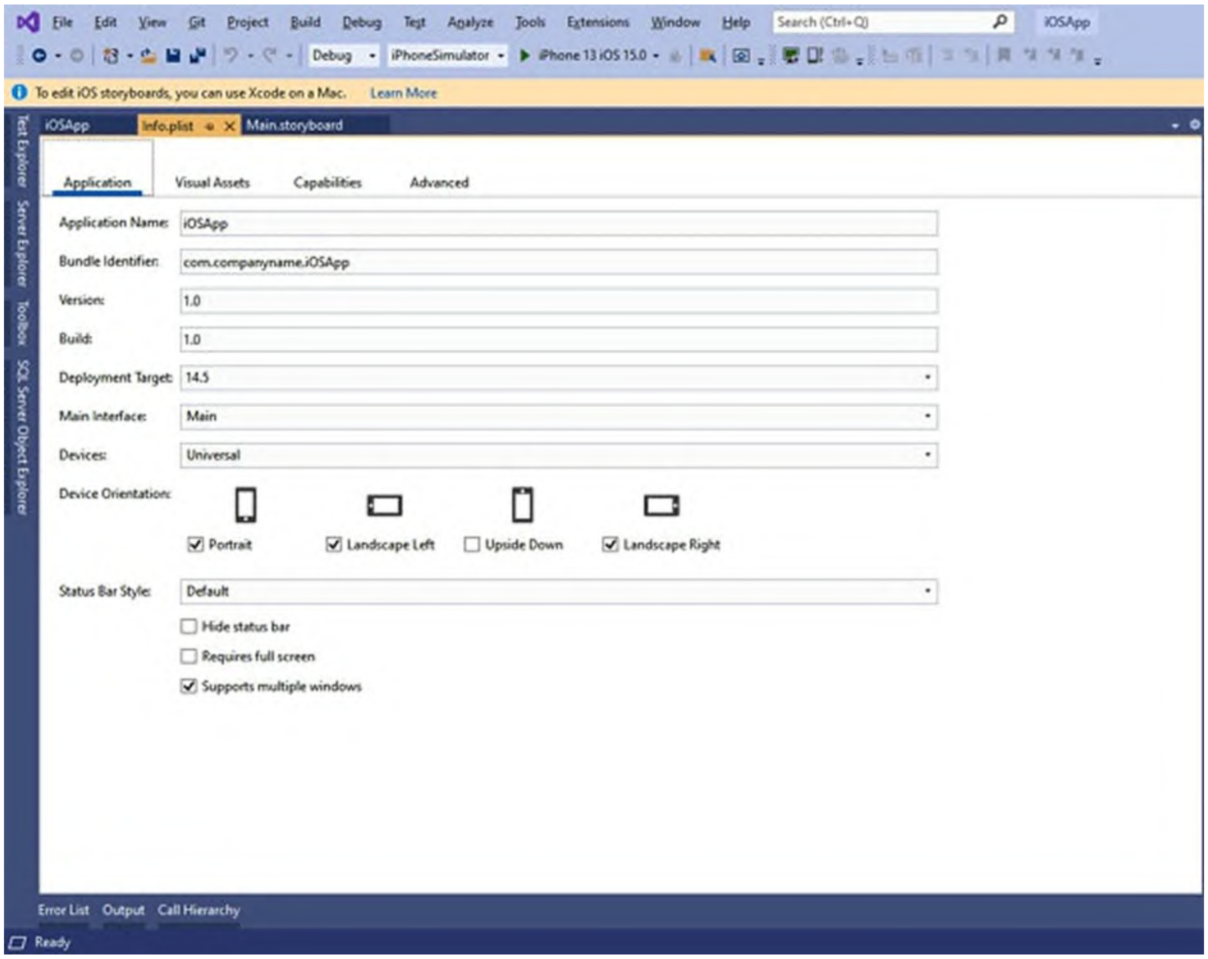
Additional Arguments:

***Figure 5.18:*** *Managing provisioning profiles for iOS*

If the **`Manual Provisioning`** option is selected, you will see a profile called **Developer**, which is used to debug the application. This is the only option you can use if you do not have a paid Apple subscription. If you have one, you will also see additional profiles that can be used to publish the app, and you can also take advantage of the **`Automatic Provisioning`** option so that Visual Studio does the job of configuring the account for you. The only thing you will need to do is click on Add **`Account`**, provide your Apple ID credentials, and wait for the information to be downloaded and connected. In summary, a free developer account is offered by Apple, but it can only be used for development and debugging against simulators and physical devices, not for distributing or publishing the app. A provisioning profile is required for physical devices, not for simulators, and Visual Studio generates them for you.

## Configuring App Package options

Like for Android, where you have the app manifest, on iOS you can also configure some important project properties. This is accomplished by editing the **`Info.plist`** file. If you double-click on it, you will see the editor shown in *Figure 5.19*:



To edit iOS storyboards, you can use Xcode on a Mac. Learn More

Application | Visual Assets | Capabilities | Advanced

Application Name: iOSApp
Bundle Identifier: com.companyname.iOSApp
Version: 1.0
Build: 1.0
Deployment Target: 14.5
Main Interface: Main
Devices: Universal
Device Orientation: ☑ Portrait ☑ Landscape Left ☐ Upside Down ☑ Landscape Right
Status Bar Style: Default
☐ Hide status bar
☐ Requires full screen
☑ Supports multiple windows

***Figure 5.19:*** *Editing the application properties*

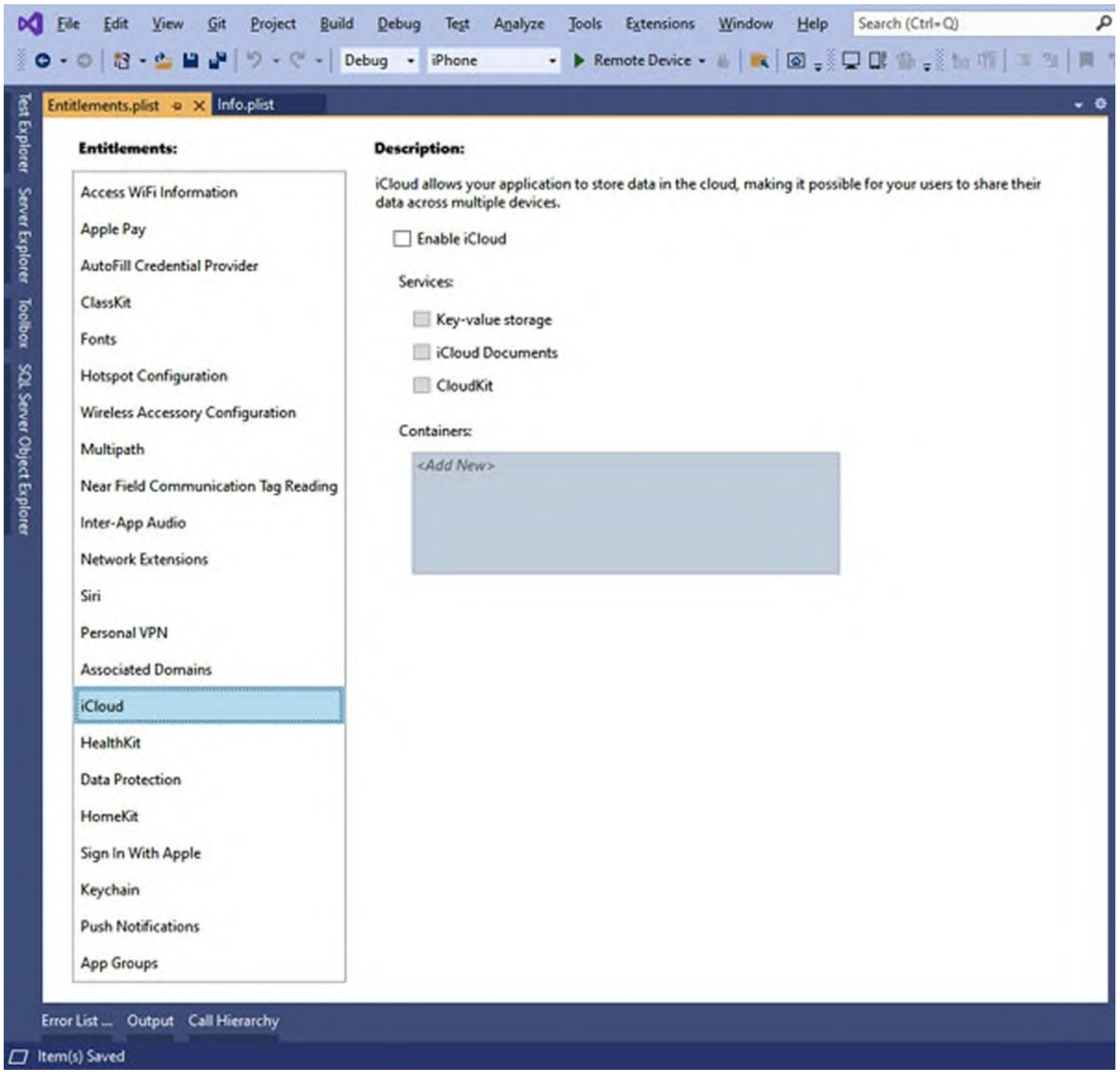
The **`Application Name`** field allows you to specify the name of the app as it will appear on the device. **`Bundle Identifier`** represents a unique identifier across the entire App Store, and it is made of the com. prefix, followed by the company name and an app identifier. Note how you can also change the target devices you selected when creating the project, and how you can specify the supported device orientations. You are not really creating Xamarin.iOS project, so you will work with Xamarin.Forms. Hence, it is recommended that you do not change the **`Main Interface`** option value, which actually points to the **`.storyboard`** file used as the root page. The **`Hide`** status bar and **Requires full screen** options allow for hiding the **iOS status** bar (clock, battery status, network) and for taking all the available screen space, respectively. The latter is particularly useful with the most recent devices that do not have the physical home button.

**It is not possible to summarize all the capabilities and integration options in this general-purpose book. So, you only get hints about the possibilities here; you are encouraged to look at the official documentation (https://docs.microsoft.com/en-us/xamarin/ios) for more information.**

Unlike Android, you do not need to set permissions that the app needs to ask to the user. These will be handled directly in code. The **Info.plist** editor provides three other tabs:

- **Visual Assets**: This is where you can provide an icon for the app, a startup image, and two icons for iTunes.
- **Capabilities**: Here, you can see if the app needs to integrate with services like the Game Center, Maps, and background modes. This is not covered in detail, so take a look at the documentation (**https://docs.microsoft.com/en-us/xamarin/ios/deploy-test/provisioning/capabilities**).
- **Advanced**: In this tab, you can specify if the app can open specific file types or URLs.

In addition, you can further integrate the application with more Apple services like Siri and iCloud. This can be accomplished by editing the **Entitlements.plist** file. *Figure 5.20* shows an example of how it looks when editing **iCloud** settings.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Debug
iPhone
Remote Device
Entitlements.plist
Info.plist
Test Explorer
Server Explorer
Toolbox
SQL Server Object Explorer
Error List ... Output Call Hierarchy
Item(s) Saved

**Entitlements:**

Access WiFi Information
Apple Pay
AutoFill Credential Provider
ClassKit
Fonts
Hotspot Configuration
Wireless Accessory Configuration
Multipath
Near Field Communication Tag Reading
Inter-App Audio
Network Extensions
Siri
Personal VPN
Associated Domains
iCloud
HealthKit
Data Protection
HomeKit
Sign In With Apple
Keychain
Push Notifications
App Groups

**Description:**

iCloud allows your application to store data in the cloud, making it possible for your users to share their data across multiple devices.

☐ Enable iCloud

Services:

☐ Key-value storage
☐ iCloud Documents
☐ CloudKit

Containers:

<Add New>

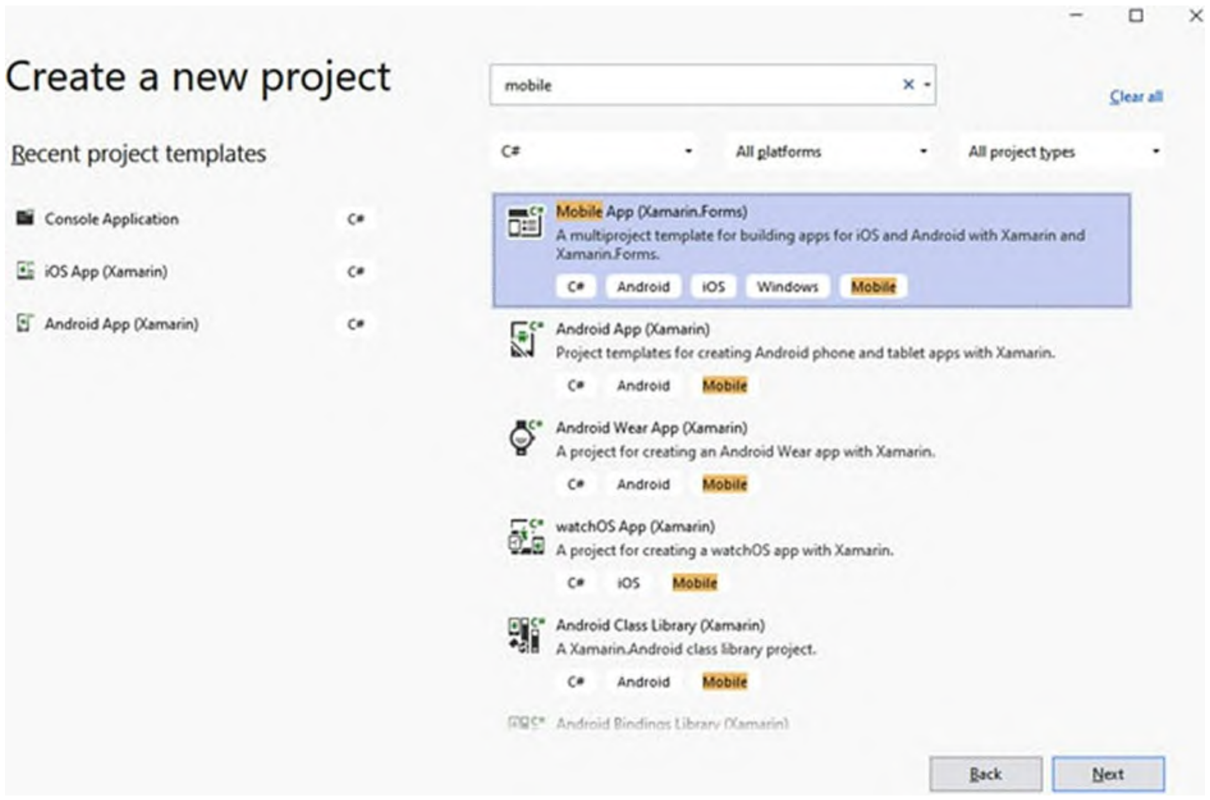
***Figure 5.20:*** *Editing integration options with Apple services*

Remember that both the **`Info.plist`** and **`Entitlements.plist`** files are not a prerogative of Xamarin.iOS; rather, they are Apple file formats and can be also edited in XCode if necessary. Now that you also have a general knowledge about Xamarin.iOS, it is time to move on to what the book focuses on: Xamarin.Forms.

**So far, you have been working with Android and iOS projects in C#, but both are individual projects, and there is no cross-platform development. This will start in the next section.**

## Cross-platform projects with Xamarin.Forms

Xamarin.Forms is the latest addition to the Xamarin code base; it was added a few years ago. The goal of Xamarin.Forms is to make it possible to develop cross-platform projects in C#, sharing as much code as possible. In particular, it lets you share code for user interface elements that are available to all platforms and logic that works on all platforms. As you will learn throughout the book, you can empower your projects with platform-specific features, which is actually a very common requirement. Xamarin.Forms is what you will use in this book, so the concepts explained in this section are extremely important. In the **`Create a new project`** dialog, locate the **`Mobile App (Xamarin.Forms)`** project template, for example, typing mobile in the search list to filter the results, as shown in *Figure 5.21*:



Mobile App (Xamarin.Forms)
A multiproject template for building apps for iOS and Android with Xamarin and Xamarin.Forms.
C# Android iOS Windows Mobile

Android App (Xamarin)
Project templates for creating Android phone and tablet apps with Xamarin.
C# Android Mobile

Android Wear App (Xamarin)
A project for creating an Android Wear app with Xamarin.
C# Android Mobile

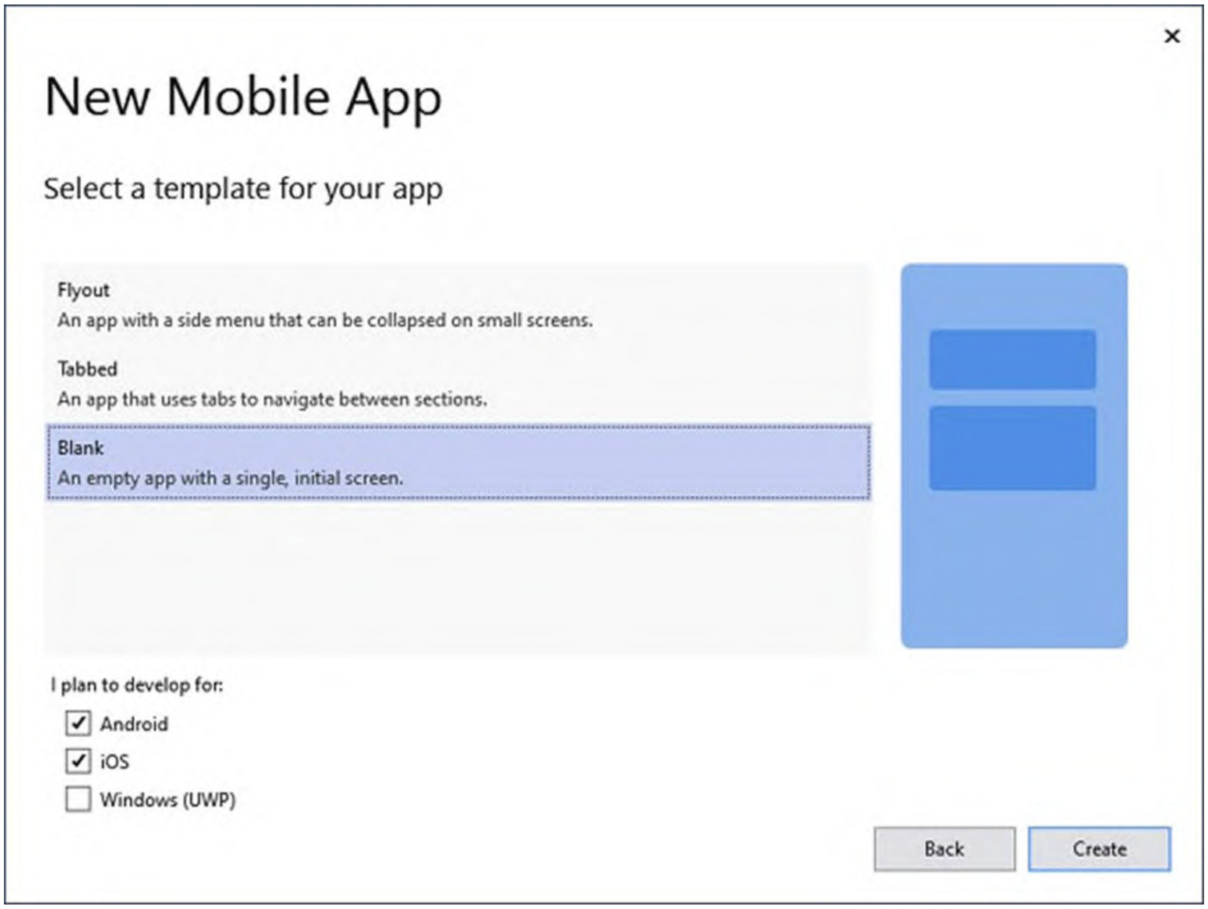
watchOS App (Xamarin)
A project for creating a watchOS app with Xamarin.
C# iOS Mobile

Android Class Library (Xamarin)
A Xamarin.Android class library project.
C# Android Mobile

Android Bindings Library (Xamarin)

***Figure 5.21:*** *Creating a Xamarin.Forms project*

When you click on **`Next`**, you will be asked to enter a project name. For the current example, enter **`MobileApp`** and then click on **`Create`**. At this point, the **`New Mobile App`** dialog appears and here, you can specify a template for your app, as shown in *Figure 5.22*:



New Mobile App

Select a template for your app

Flyout
An app with a side menu that can be collapsed on small screens.

Tabbed
An app that uses tabs to navigate between sections.

Blank
An empty app with a single, initial screen.

I plan to develop for:

[✓] Android
[✓] iOS
[ ] Windows (UWP)

***Figure 5.22:*** *Selecting an app template*

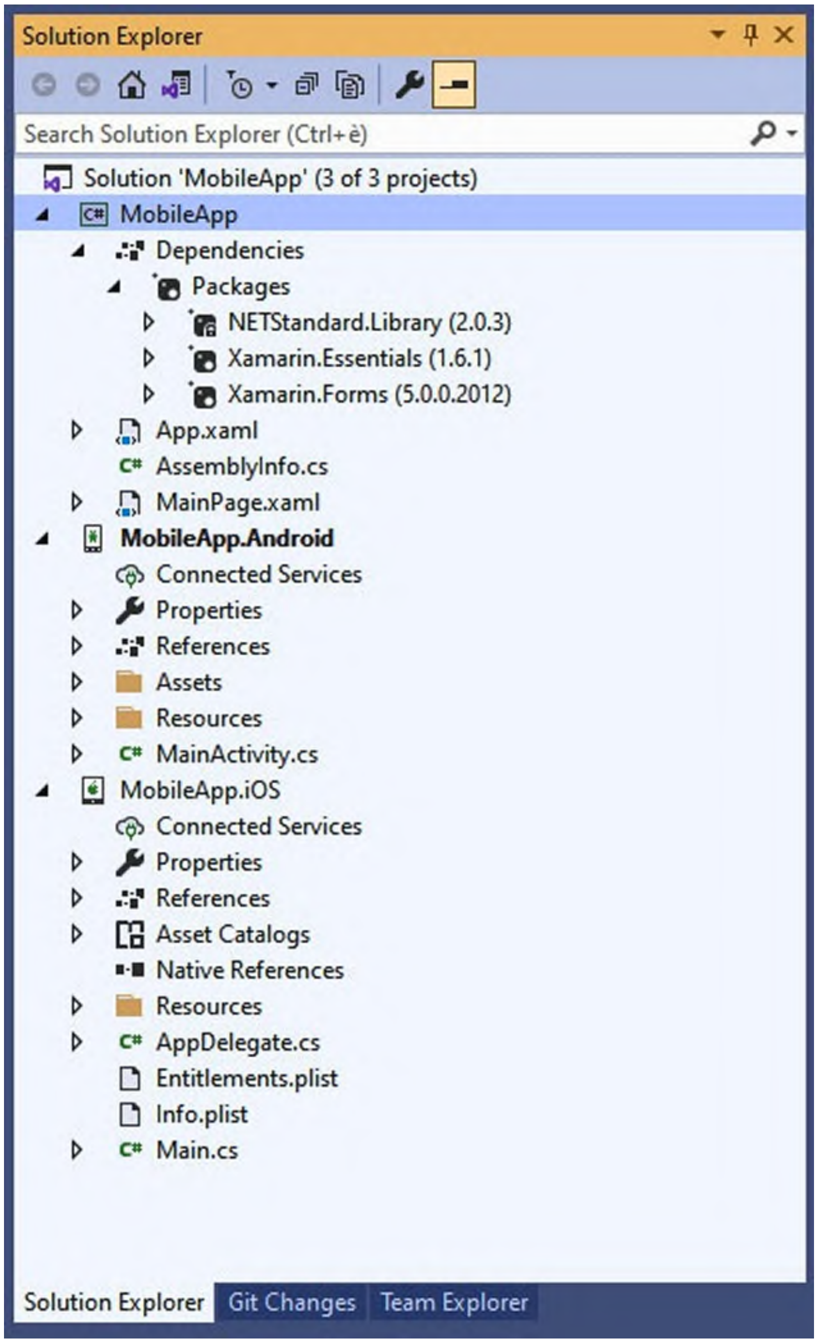
There are three templates:

- **`Flyout`**, which creates a basic infrastructure with a collapsible side menu.
- **`Tabbed`**, which creates a basic infrastructure based on tabs to enable page navigation.
- **`Blank`**, which creates an empty project with one page.

For now, select **`Blank`**. Then, you can find the list of supported platforms that you wish to target. As mentioned in *Chapter 1, The Importance of Mobile App Development*, and *Chapter 2, Xamarin and Microsoft in the Mobile App Market*, this book only focuses on Android and iOS, so make sure that these two platforms are selected. Finally, click on **`Create`**. After a few seconds, the project will be ready in Visual Studio.

## Understanding the project structure

If you look at Solution Explorer (see *Figure 5.23*), you can see how a Xamarin.Forms solution is made of three projects (they would have been four if you had also selected UWP as a target):



```
Solution 'MobileApp' (3 of 3 projects)
  MobileApp
    Dependencies
      Packages
        NETStandard.Library (2.0.3)
        Xamarin.Essentials (1.6.1)
        Xamarin.Forms (5.0.0.2012)
    App.xaml
    AssemblyInfo.cs
    MainPage.xaml
  MobileApp.Android
    Connected Services
    Properties
    References
    Assets
    Resources
    MainActivity.cs
  MobileApp.iOS
    Connected Services
    Properties
    References
    Asset Catalogs
    Native References
    Resources
    AppDelegate.cs
    Entitlements.plist
    Info.plist
    Main.cs
```

***Figure 5.23:*** *The structure of a Xamarin.Forms solution*

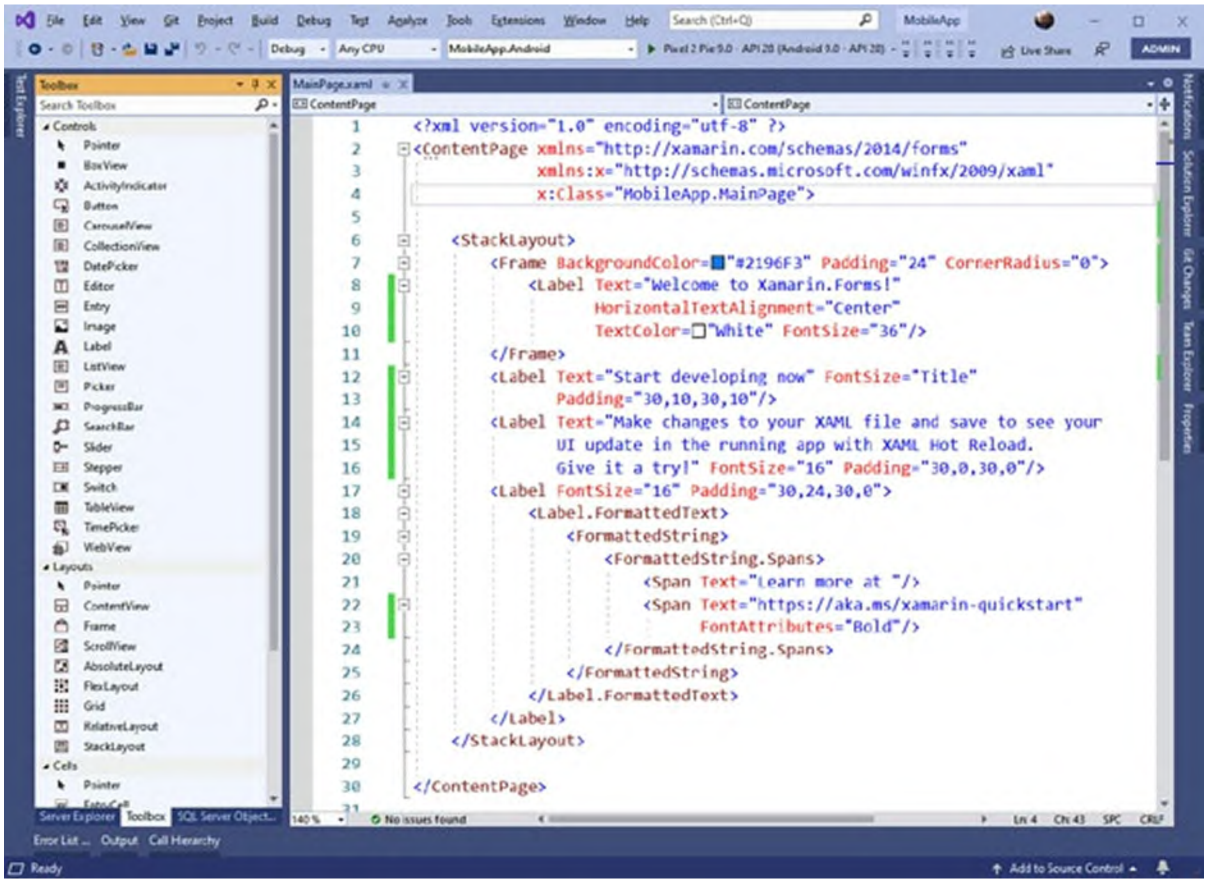
The **`MobileApp`** project is also referred to as shared project because it is the real place where you write code, logic, and the user interface. Its main dependencies are the Xamarin.Forms library and the .NET Standard Library, whereas Xamarin.Essentials is an additional library that is added to support common tasks. Then, you can find two native projects, **`MobileApp.Android`** and **`MobileApp.iOS`**, which have the same structure and behavior you learned previously in this chapter, and both have a reference to the shared project. The shared project can be thought of as a connection point between native projects and the Xamarin libraries, so they are still the place where you customize app information, as you learned in the previous sections.

## Introducing XAML, App.xaml, and MainPage.xaml

In Solution Explorer, you will also see two files, **`App.xaml`** and **`MainPage.xaml`**. The **`App.xaml`** file contains resources that are globally available in the app and allows for managing the application lifecycle events in its code-behind C# file, called **`App.xaml.cs`**, that you can see if you expand the **`App.xaml`** node. On the other hand, **`MainPage.xaml`** represents the auto-generated root page. It contains the visual elements of the user interface and navigation where required. Its code-behind file, **`MainPage.xaml.cs`**, is used to handle user actions over the user interface and to manage the page lifecycle. Especially in the next two chapters, you will learn how to work with multiple pages. The **`.xaml`** extension means that the file contains XAML markup code. XAML stands for eXtensible Application Markup Language, a markup language that derives from XML and that allows for creating the user interface.

**As a general rule, a C# code-behind file exists for every XAML file. There can be exceptions, but this is the most common situation.**

If you click on **`MainPage.xaml`**, you will see some sample XAML markup generated by Visual Studio in the code editor (see *Figure 5.24*):



```xml
<?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="MobileApp.MainPage">

    <StackLayout>
        <Frame BackgroundColor="#2196F3" Padding="24" CornerRadius="0">
            <Label Text="Welcome to Xamarin.Forms!"
                   HorizontalTextAlignment="Center"
                   TextColor="White" FontSize="36"/>
        </Frame>
        <Label Text="Start developing now" FontSize="Title"
               Padding="30,10,30,10"/>
        <Label Text="Make changes to your XAML file and save to see your
               UI update in the running app with XAML Hot Reload.
               Give it a try!" FontSize="16" Padding="30,0,30,0"/>
        <Label FontSize="16" Padding="30,24,30,0">
            <Label.FormattedText>
                <FormattedString>
                    <FormattedString.Spans>
                        <Span Text="Learn more at "/>
                        <Span Text="https://aka.ms/xamarin-quickstart"
                              FontAttributes="Bold"/>
                    </FormattedString.Spans>
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>

</ContentPage>
```

***Figure 5.24:*** *Defining the user interface via XAML*

XAML is based on the XML syntax. More specifically, an XML node represents a visual element, whereas XML attributes represent properties of a visual element. In the sample code, there are the following relevant elements:

- **`ContentPage`** is an object that represents an individual page.
- **`xmlns`** tags can be compared to C# namespaces and allow for importing objects from a library or built-in schema, like in the case of Xamarin types.
- **`StackLayout`** is a layout, which can be thought of as a container of visual elements.
- **`Frame`** and **`Label`** are visual elements, normally referred to as views in mobile app development terminology, or as controls in .NET and C# terminology.

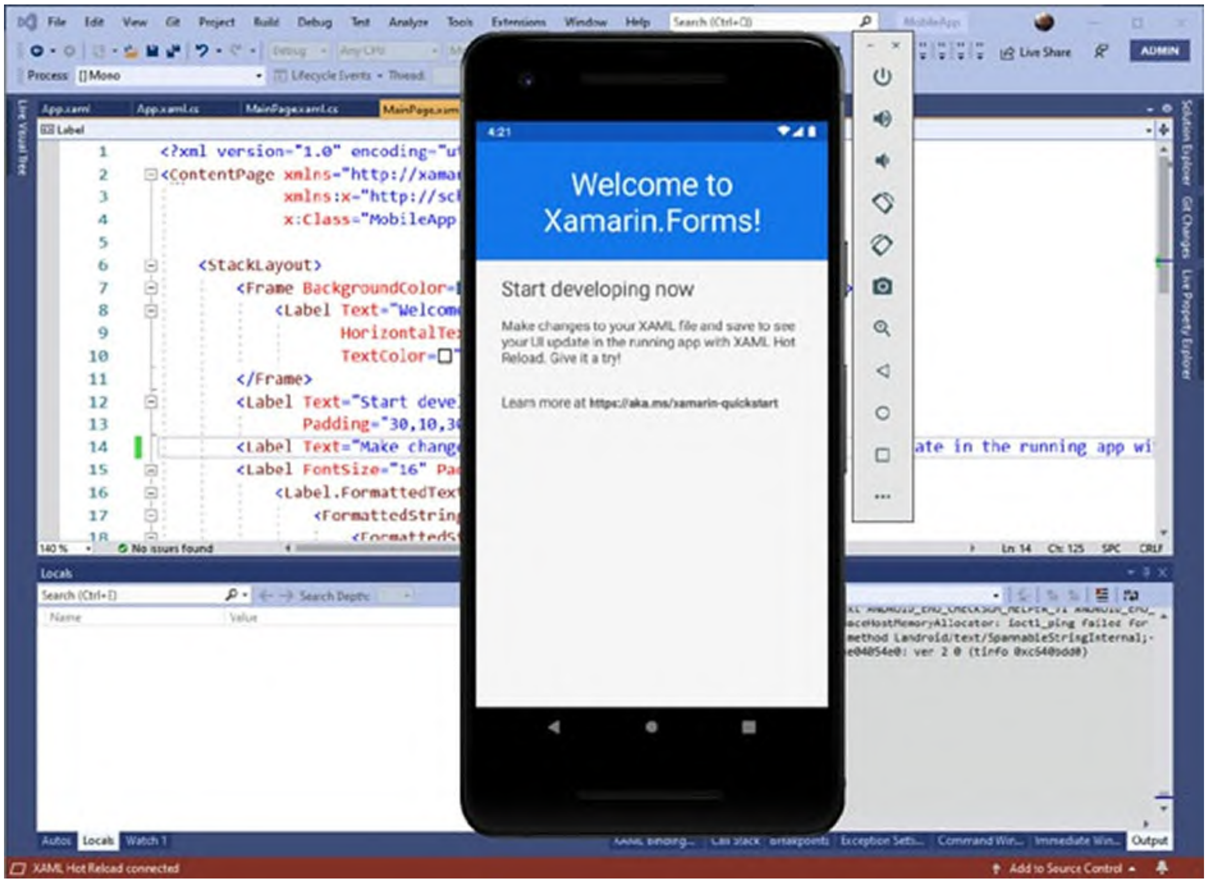
All the layouts and views will be discussed in the upcoming chapters; for now, you just need to focus on how XAML allows for defining the user interface. *Figure 5.24* also shows the **`ToolBox`** window, where you can find a list of supported visual elements that you can drag onto the XAML code. You could also write the user interface entirely in C#, and sometimes, this is useful when you need to create visual elements at runtime. However, XAML has the following benefits:

- It makes it simpler to define the user interface with a hierarchical approach.
- It allows for completely separating the user interface definition (declarative code) from the actions (imperative code).

Separation also means that professional designers can use specific tools to edit the XAML markup without touching the imperative code. Previously, Visual Studio included a tool called Xamarin.Forms Previewer, which enabled getting a visual representation of the XAML code within device simulator. However, this tool has been deprecated in favor of a new one called Xamarin Hot Reload. With this approach, you first write your XAML and C# and then run the app for debugging and make changes in the code at runtime. Hot Reload will be shown as we move further. Luckily enough, IntelliSense for the XAML editor is very powerful, so declaring the user interface will be quite simple. The C# code-behind for the page is very simple; it only contains a constructor that invokes a method called **`InitializeComponent`**, which is inherited from the base Page class. **`App.xaml`** is a container of resources that are shared across the app. Resources in Xamarin.Forms are a specific topic, which is discussed in *Chapter 9: Resources and Data Binding*. The **`App.xaml.cs`** file usually contains any code that needs to be initialized when the app starts up. In addition, it allows for handling the application lifecycle events (**`OnStart`**, **`OnSleep`**, **`OnResume`**). These are all quick hints that will be described in detail in *Chapter 11, Managing the Application Lifecycle*.

## Running and debugging apps

When you have written some code and want to run your app, you first need to select a startup project, which can either be the Android project or the iOS one. For example, in Solution Explorer, right-click on the **`MobileApp.Android`** project and then select **`Set as Startup Project`**. The name of the project will be highlighted in bold. Now, you will be able to run the app as you already know, for example, in the emulator of choice. [Figure 5.25](#) shows the sample app running in the Android emulator. The reason why you see Visual Studio behind the emulator is to highlight the fact that you can make changes to your XAML code while debugging, and these will be immediately reflected into the user interface once you save the code file (Hot Reload):



```
<?xml version="1.0" encoding="u
<ContentPage xmlns="http://xama
             xmlns:x="http://sc
             x:Class="MobileApp

    <StackLayout>
        <Frame BackgroundColor=
            <Label Text="Welcom
                   HorizontalTe
                   TextColor=□"
        </Frame>
        <Label Text="Start deve
               Padding="30,10,3
        <Label Text="Make chang                              ate in the running app wi
        <Label FontSize="16" Pa
            <Label.FormattedTex
                <FormattedStrin
                    <FormattedS
```

```
aceHostMemoryAllocator: ioctl_ping failed for
method Landroid/text/SpannableStringInternal;-
e04054e0: ver 2 0 (tinfo 0xc540bdd0)
```

4:21
Welcome to Xamarin.Forms!
Start developing now
Make changes to your XAML file and save to see your UI update in the running app with XAML Hot Reload. Give it a try!
Learn more at https://aka.ms/xamarin-quickstart
App.xaml
App.xaml.cs
MainPage.xaml.cs
MainPage.xaml
Process: [] Mono
Lifecycle Events
Live Share
ADMIN
Locals
Name
Value
No issues found
Ln 14
Ch 125
SPC
CRLF
Autos
Locals
Watch 1
Output
XAML Hot Reload connected
Add to Source Control

*Figure 5.25:* Running a Xamarin.Forms project

Remember the following things:

- You will write cross-platform code in the shared project. This is what you will do most of the time.
- You will manage native resources (build options, application artwork, etc.) directly in each native project.

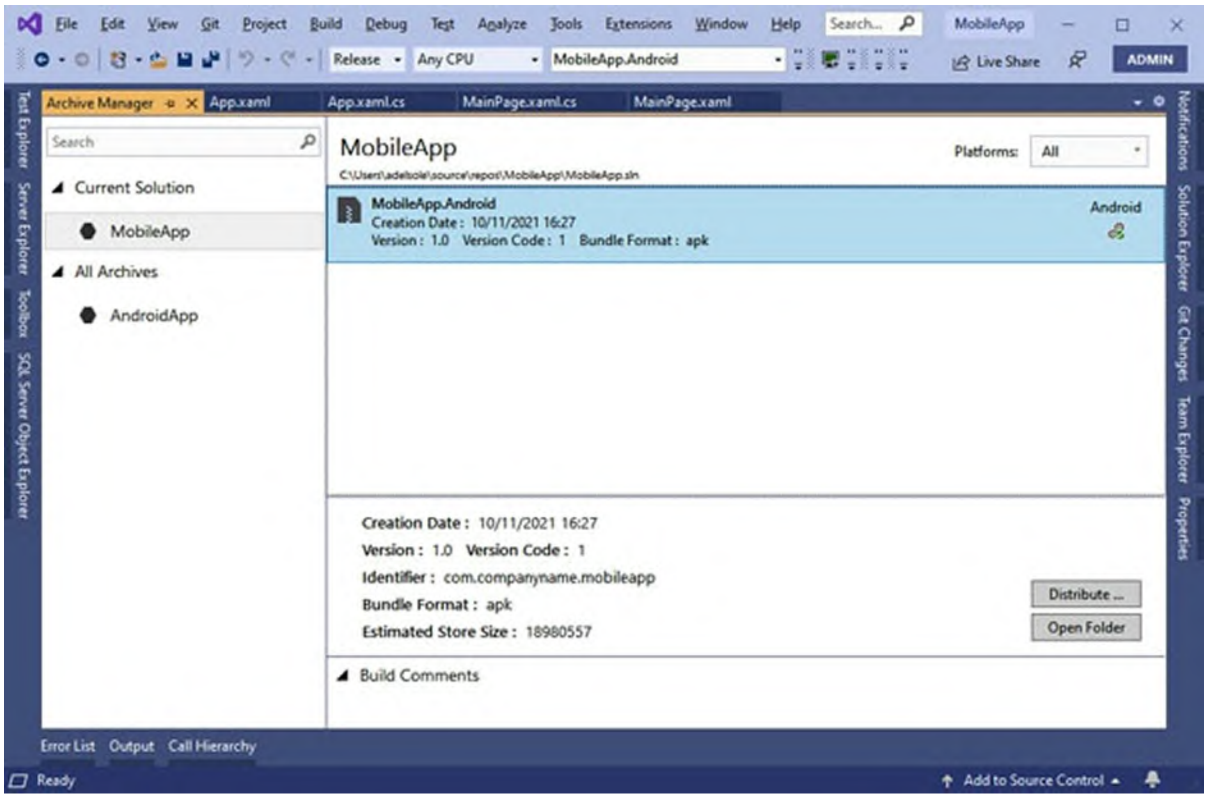
All these general concepts will be covered often in the upcoming chapters, so do not be scared if something is not clear.

## Preparing apps for publication

When you complete the development and testing of your apps, you likely want to publish them. Publishing apps to Google Play and the Apple App Store requires paid subscriptions, and the details are not covered in this book. However, you will now learn how to prepare an app for distribution.

## Preparing Android packages

Let's consider Android first. Select **`Release configuration`**, and then right-click on the project name in **`Solution Explorer`** and select **`Archive`**. Visual Studio will prepare a so-called archive, whose summary is shown in *Figure 5.26*:



MobileApp
C:\Users\adebole\source\repos\MobileApp\MobileApp.sln

MobileApp.Android
Creation Date : 10/11/2021 16:27
Version : 1.0 Version Code : 1 Bundle Format : apk
Android

Creation Date : 10/11/2021 16:27
Version : 1.0 Version Code : 1
Identifier : com.companyname.mobileapp
Bundle Format : apk
Estimated Store Size : 18980557

***Figure 5.26:*** *Archiving a package for publication*

The **`Archive Manager`** tool that you see in *Figure 5.26* also contains the list of previously archived packages. The next step is to click on **`Distribute`**. You will be asked which channel you want to use, as shown in *Figure 5.27*:

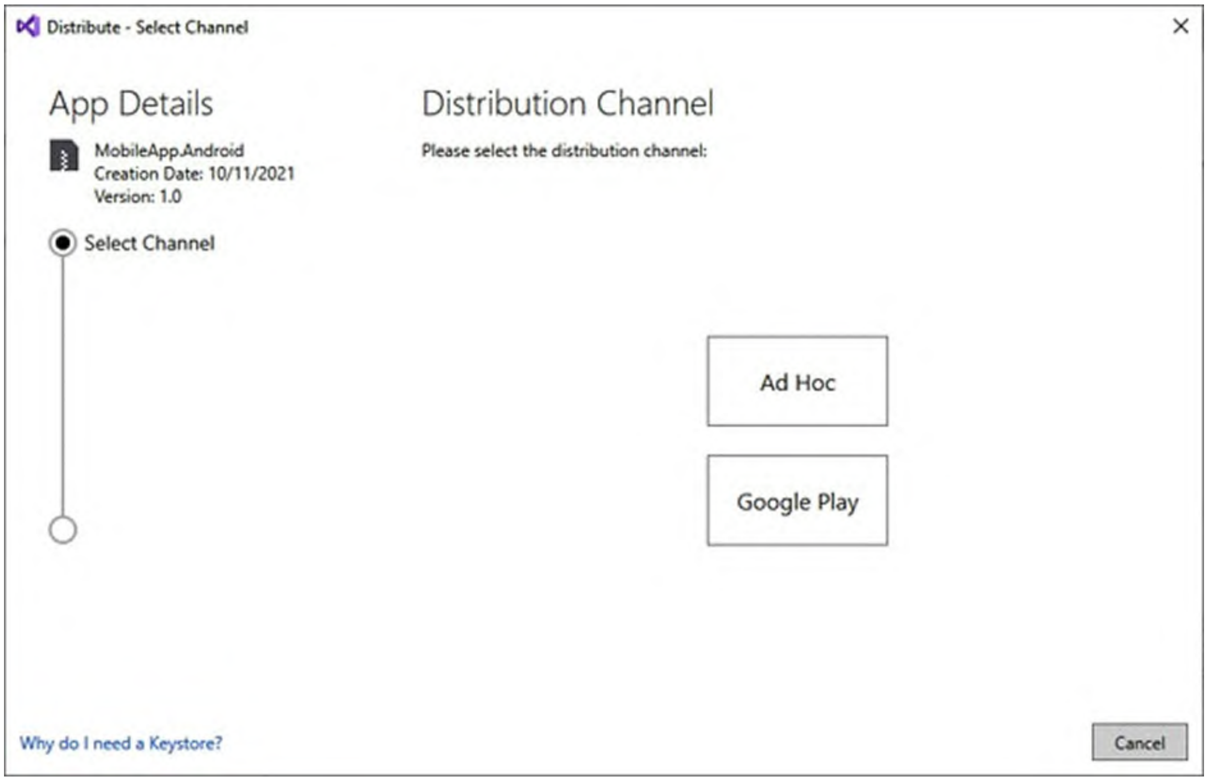
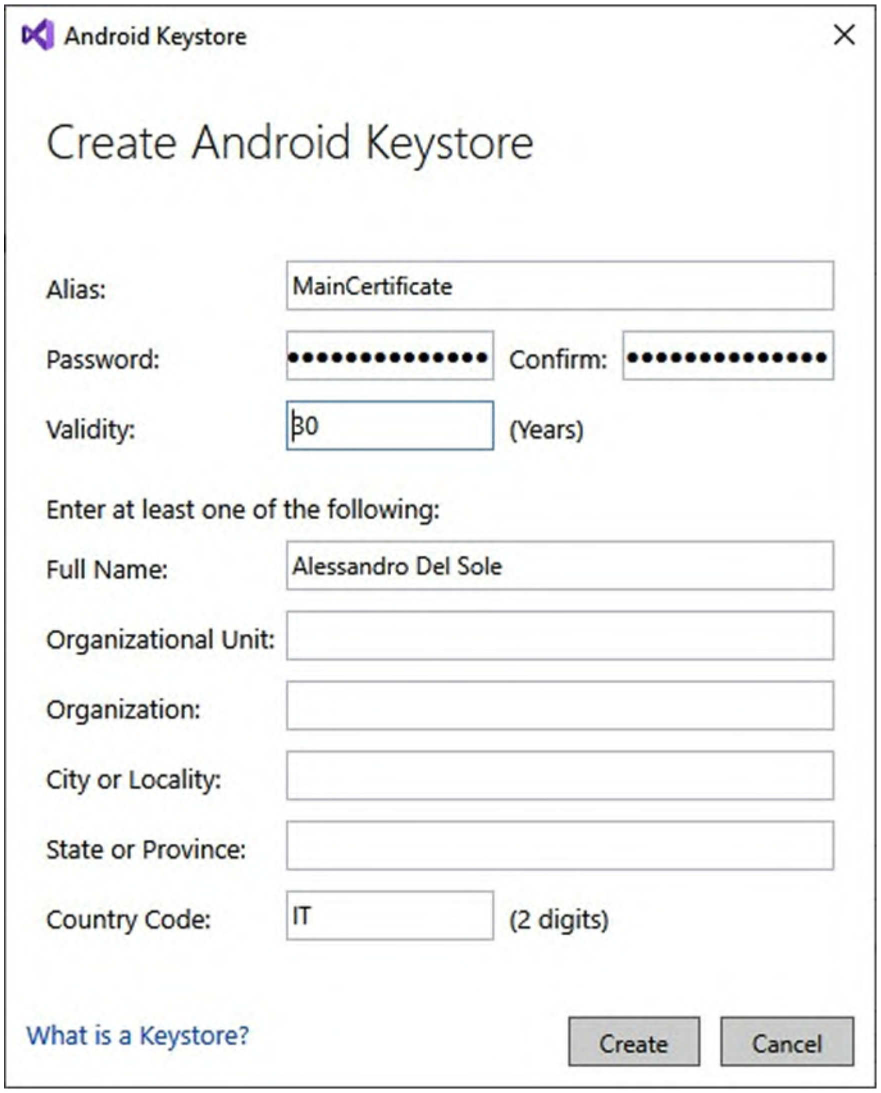

Distribute - Select Channel
App Details
MobileApp.Android
Creation Date: 10/11/2021
Version: 1.0
Select Channel
Distribution Channel
Please select the distribution channel:
Ad Hoc
Google Play
Why do I need a Keystore?
Cancel

***Figure 5.27:*** *Selecting a distribution channel*

The **`Ad Hoc`** distribution channel allows you to create a signed **`.apk`** file that you can freely share with others. If you select **`Google Play`** instead, Visual Studio will ask for your Google Play credentials and guide you through the process of publishing the app to the store. However, you can always go for the **`Ad Hoc`** option and manually upload the generated **`.apk`** to Google Play at a later stage. Click on **`Ad Hoc`**. You will be asked to **`Create an Android keystore`**, as shown in *Figure 5.28*:



| | |
|---|---|
| Alias: | MainCertificate |
| Password: | •••••••••••••• |
| Confirm: | •••••••••••••• |
| Validity: | 30 (Years) |

Enter at least one of the following:

| | |
|---|---|
| Full Name: | Alessandro Del Sole |
| Organizational Unit: | |
| Organization: | |
| City or Locality: | |
| State or Province: | |
| Country Code: | IT (2 digits) |

***Figure 5.28:*** *Creating an Android keystore*

This is a certificate that you will use to sign the app package. Mandatory fields are **`Alias`**, **`Password`** and **`Full Name`**. The alias is just a friendly name for your records. Click on **`Create when ready`**, and then, in the **`Archive Manager`**, click on **`Distribute`**. When the generation of the **`.apk`** file is completed, you will be able to click on the **`Open Folder`** button to see the file in Windows Explorer. You can then share your app package with other people or manually upload it to Google Play.

## Preparing iOS packages

For iOS, things work quite similarly. You will still select the **`Release configuration`** for a project, but you will need to select the iPhone flavor and a physical device. In addition, you will still need to be connected to a Mac. You will then right-click on the project name in **Solution Explorer** and select **Archive**. Visual Studio will search for an appropriate provisioning profile on the Mac. If not found, you can create one by following these steps:

1. Open Xcode on the Mac
2. Create a new Swift project with any of the templates.
3. When Xcode asks you to specify a bundle identifier, enter exactly the same identifier you added to the Xamarin.iOS project.
4. Select a target device (either simulator or physical) and start the application via the **Start** button on the toolbar.

These steps will generate a provisioning profile on the Mac side. When done, go back to Visual Studio and repeat the previous steps. Remember that the generated **.ipa** file is located on the Mac, so you will need to pick it up from there. Apple allows you to distribute apps to internal testers via the TestFlight tool before you publish them to the regular App Store. TestFlight cannot be covered here, so it is recommended that you read the official documentation (**https://developer.apple.com/testflight/**).

## Conclusion

In this chapter, you learned how native projects are made and why they are so important inside a Xamarin.Forms solution. You also learned how to set up the development environment to work with cross-platform projects. You acquired all the necessary basic knowledge to start developing mobile apps with Xamarin.Forms. Starting with the next chapter, you will seriously start writing code, and you will learn many more concepts about Xamarin.Forms and mobile app development in general.

## Points to remember

- With Xamarin.Forms, you share the user interface and the logic across projects.
- You set up app information in the native projects' properties.
- In Xamarin.Forms, the user interface is usually written with XAML markup.

## Key terms

- **App manifest**: A file that specifies properties of an app.
- **XAML**: Markup language you use to define the user interface with a declarative approach.
- **Imperative code**: C# code you use to implement actions for your app.
- **Archive**: A signed app package ready for distribution.

# CHAPTER 6

# Organizing the User Interface with Layouts

## Introduction

You cannot predict the mobile devices your apps will work on; they can be smartphones, tablets, and even desktop computers. This means many possible screen sizes and form factors. Applications should also work with both landscape and portrait orientations. Based on these considerations, the user interface in mobile apps must be adaptive. This means that visual elements will automatically and dynamically resize to adapt to the device, screen size, and form factor. To accomplish this, Xamarin.Forms provides layouts, which is the topic of this chapter.

**Across the book, you will often find terms like layout, view, control, visual element. A layout, as you learn in this chapter, can contain one or more views. A view is an individual piece of the user interface (for example, a text box, a button, and so on) and the view term is often used in native Android and iOS development. Control has the same meaning as view, but it is more often used in the Microsoft terminology. A visual element can be any element of the user interface, including layouts and views. Control and view are used in this book interchangeably.**

## Structure

In this chapter, we will cover the following topics:

- Understanding the concept of layout
- Organizing the user interface with Xamarin.Forms layouts
- Styling the user interface with cascading style sheets

## Objectives

After completing this chapter, you will be able organize visual elements in a dynamic way, and you will be able to decide which layout suits better your needs in a specific scenario. The chapter comes with a companion Xamarin.Forms solution that you can open in Visual Studio to quickly follow the examples. The shared project contains several pages, each with an example about a specific layout, with self-explanatory name. The startup page is assigned in the **`App.xaml.cs`** file, passing a page instance to the **`MainPage`** property, as follows:

```
 MainPage = new StackLayoutExample();
```

The preceding line shows an example of how to run the **`StackLayoutExample`** page.

## Understanding the concept of layout

The user interface of your apps should be dynamic in most cases. This means that it should be automatically adapt to the device type, screen size, and user settings like large fonts. For this reason, the visual elements you add to the user interface should never have a fixed size or position, except in situations where fixed positioning provides the proper user experience. Xamarin.Forms makes it possible to implement dynamic user interface by arranging controls inside containers known as layouts or panels. This chapter provides a detailed description of all the available layouts, and it explains how to add controls in a dynamic approach. In addition, you will also learn that the user interface in Xamarin.Forms is built in a hierarchical way, which means that one layout can contain one or more nested layouts and create sophisticated user interfaces. *Table 6.1* describes available layouts in Xamarin.Forms.

| **Layout** | **Description** |
|---|---|
| **`StackLayout`** | Visual elements are placed one after the other, either horizontally (from left to right) or vertically (from top to bottom). |
| **`FlexLayout`** | Like the `StackLayout`, but visual elements are wrapped to the next row or column if there is not enough space on the page. |
| **`Grid`** | Visual elements are positioned inside rows and columns of a virtual table. |
| **`AbsoluteLayout`** | Allows for placing the child element at a specified position. |
|  |  |

| | |
|---|---|
| **`RelativeLayout`** | Places child elements in a position that depends on constraints established by the relationship with other elements. |
| **`ScrollView`** | Implements scrolling capabilities over the child visual elements. |
| **`Frame`** | Adds colored border, round corners, and drop shadow to the child visual element. |
| **`ContentView`** | Allows form implementing custom, reusable XAML views. |

***Table 6.1:*** *Layouts in Xamarin.Forms*

In Xamarin.Forms, there is usually at least one root layout in each page, which is declared in the page's **`Content`** property. Such a root layout typically contains the hierarchy of visual elements in the page, which can also include other layouts.

## Alignment and spacing options

In Xamarin.Forms, every visual element can be aligned horizontally or vertically in the user interface by setting the **`HorizontalOptions`** and **`VerticalOptions`** properties, respectively. Allowed values are exposed by the **`LayoutOptions`** structure and are listed in *Table 6.2*. For example, if you only have the root layout in a page, you can assign **`VerticalOptions`** with **`FillAndExpand`** so that it will fill all the available space:

| Alignment | Description |
|---|---|
| **`Center`** | The visual element is aligned at the center. |
| **`CenterAndExpand`** | The visual element is aligned at the center and fills all the available space. |
| **`Start`** | If assigned to `HorizontalOptions`, the visual element is aligned at the left. If assigned to `VerticalOptions`, the visual element is aligned at the top. |
| **`StartAndExpand`** | If assigned to `HorizontalOptions`, the visual element is aligned at the left. If assigned to `VerticalOptions`, the visual element is aligned at the top. In both cases, the visual element fills all the available space. |
| **`End`** | If assigned to `HorizontalOptions`, the visual element is aligned at the right. If assigned to `VerticalOptions`, the visual element is aligned at the bottom. |
| **`EndAndExpand`** | If assigned to `HorizontalOptions`, the visual element is aligned at the right. If assigned to `VerticalOptions`, the visual element is aligned at the bottom. In both cases, the visual element fills all the available |

| | |
|---|---|
| | space. |
| **`Fill`** | The visual element has no spacing around its child elements, and it does not expand. |
| **`FillAndExpand`** | The visual element has no spacing around its child elements, and it fills all the available space. |

***Table 6.2:** Alignment options*

When arranging the user interface, it is also important to properly manage spacing between visual elements and layouts, and inside individual views. This is accomplished by leveraging the following three properties: **`Padding`**, **`Spacing`**, and **`Margin`**. *Table 6.3* describes them in further detail:

| Property | Description |
|---|---|
| **`Margin`** | Of type thickness, this is the space between a visual element and its adjacent views. |
| **`Padding`** | Of type thickness, this is the space between a visual element and its child elements. |
| **`Spacing`** | Of type double, it is only available in the `StackLayout` and specifies the spacing between each child element. The default is 6. |

***Table 6.3:** Spacing options*

Properties of type thickness can have a fixed value or different values for each side. For example, consider the following assignment:

```
Margin = “10"
```

It adds the same distance to all the sides of a view from its adjacent visual elements. Instead, consider the following assignment:

```
Margin = “10,5,10,5"
```

It adds a different distance to each side of the view in the following order: left, top, right, bottom.

Spacing and distances might not be so immediate to implement, so it is recommended that you make some experiments for better, practical understanding.

## Understanding the visual tree

In all the technologies based on XAML, such as Xamarin.Forms, Windows Presentation Foundation, and Universal Windows Platform, you will often hear about a concept known as **visual tree**. The visual tree is the hierarchy of visual elements that make the user interface and their primitive elements. For example, consider the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Layouts.MainPage">
  <StackLayout Orientation="Vertical">
    <Label Text="Example" />
    <StackLayout>
    </StackLayout>
  </StackLayout>
</ContentPage>
```

This code generates a page with some visual elements inside, and its visual hierarchy is represented by the visual tree. At the top of the visual tree, there is a **`ContentPage`**, which represents an individual page. The next level of the hierarchy is a **`StackLayout`** (explained in the upcoming sections). Then, there are two objects on the same level: a **`Label`** and a **`StackLayout`**. If you have some programming experience, you know that usually visual elements also have a name to uniquely identify them. But in the preceding XAML code, no visual element has a name. Behind the scenes, the Xamarin.Forms runtime can identify and handle each visual element even if they have no names. This will be particularly useful when working with databinding in *Chapter 9, Working with Resources and Data Binding*. You will need to specify a name if you want to interact with a visual element in C# code or when you need to reference it from another visual element in XAML. You assign a name via the **`x:Name`** tag, like in the following example:

```
<Label x:Name="WelcomeLabel" />
```

This allows you to handle the Label in C#, like in the following code:

```
WelcomeLabel.Text="Example";
```

In the next few pages, you will seldom provide a name for visual elements, but you will do this more often starting with the next chapter.

## .NET objects hierarchy

In terms of .NET objects, all the layouts described in the next section derive from the **`Xamarin.Forms.Layout`** class, which exposes properties and structure that are common to each layout. The **`Layout`** class derives from **`Xamarin.Forms.View`**, a class that defines the infrastructure for layouts and views. View inherits from **`Xamarin.Forms.VisualElement`**, a class that defines the basic structure of any visual element. For example, the **`HorizontalOptions`** and **`VerticalOptions`** properties whose values were summarized in *Table 6.2* are defined in the **`View`** class. In practice, even if it is good for your knowledge, it is not necessary to remember the inheritance level of each property. You can quickly use the **`Go to Definition`** tool and the **`Object Browser`** window to dissect an object's structure. For this reason, members derived via inheritance will be highlighted only when necessary.

## Organizing the user interface

This section describes how to create a sample project, with concepts that you will use multiple times across the book, and it shows how to organize elements of the user interface with the layouts summarized in *Table 6.1*.

## Creating a Sample Project

As stated previously, the chapter comes with a companion Xamarin.Forms solution that you can open with Visual Studio to better follow the examples. However, if you wish to create a project from scratch, you can follow these steps:

- By following the lesson learned in the previous chapter, create a new Xamarin.Forms solution called **`Layouts`** for consistency with the sample solution.
- Do not edit or remove the auto generated **`MainPage.xaml`** file; it will be used later.
- For each layout discussed in the book, add a new item of type Content Page (XAML). To accomplish this, right-click on the shared project name and then click on **`Add New Item`** in the context menu.
- In the **`Add New Item`** dialog, click on the Xamarin.Forms node on the left and then select the **`Content Page (XAML)`** item template.
- Assign to the new XAML file a name that matches the discussed layout, such as **`StackLayoutExample.xaml`**, and click on **`Add`**.

This is also a useful exercise to get familiar with creating and managing Xamarin.Forms solutions.

## The StackLayout

With the **`StackLayout`** layout, child views are positioned near each other. Child views can be oriented both horizontally and vertically; the vertical orientation is the default. The following code demonstrates how to arrange order child views horizontally and vertically, and how a **`StackLayout`** can contain nested layouts:

```
<?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.StackLayoutExample">
  <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 this XAML is shown in *Figure 6.1*, where you can see both the Android and iOS simulators in action.

**All figures representing the user interface as a result of some code will show both the Android and iOS simulators so that you can have a better idea of how your code is working cross-platform. If a feature is only available in one of the systems, only the related device will be shown.**

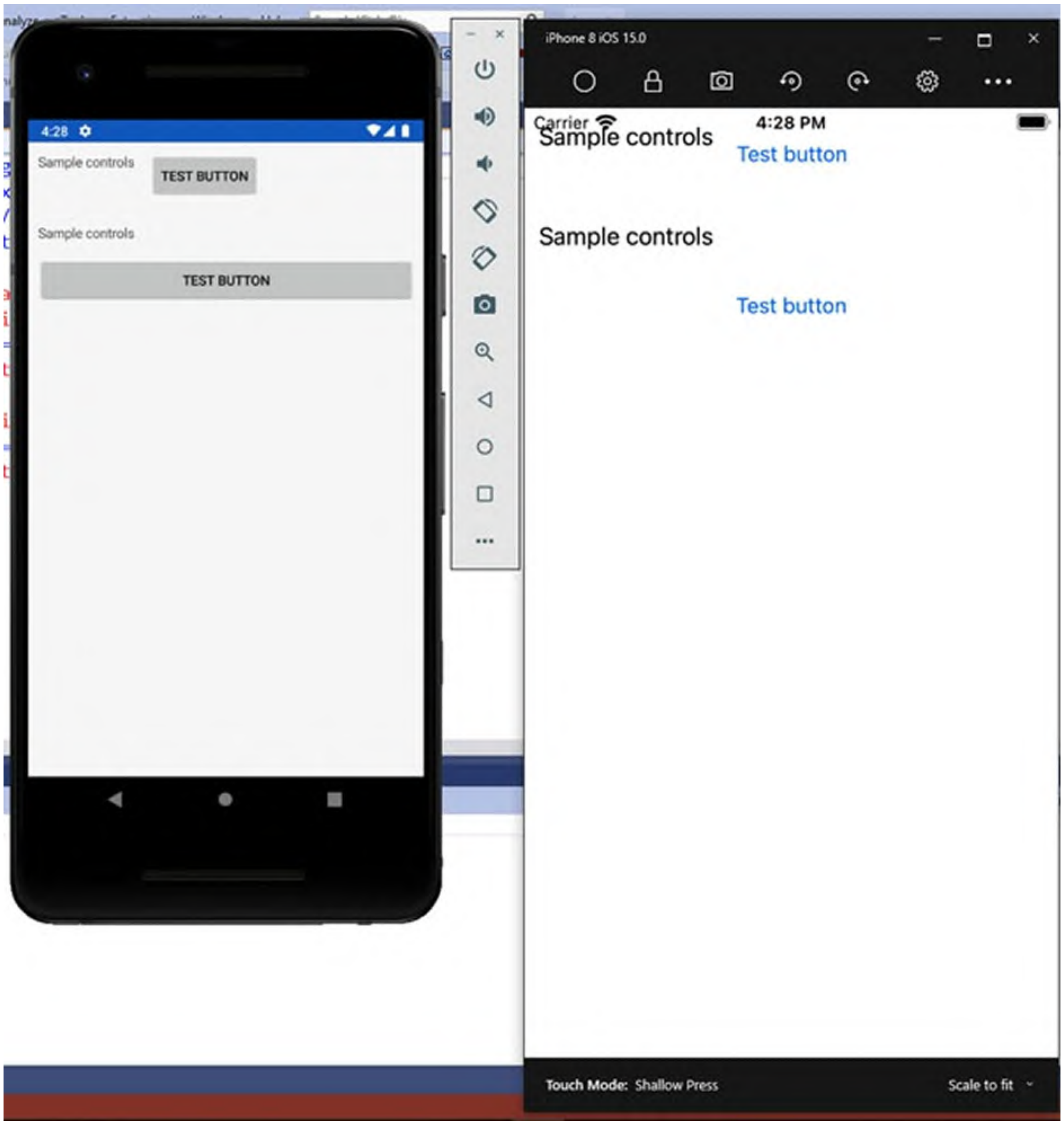


***Figure 6.1:*** *Arranging views with the StackLayout*

The **`Orientation`** property can be assigned with **`Horizontal`** or **`Vertical`**; the latter is the default. In terms of user interface dynamicity, child views of a **`StackLayout`** are proportionally resized depending on the orientation. You can also assign child views a fixed size by assigning the **`WidthRequest`** and **`HeightRequest`** properties of each child view. These represent the width and height, respectively, but it is preferable to not assign a fixed size. As mentioned in *Table 6.3*, you can also decide the spacing between child views with the **`Spacing`** property.

## The FlexLayout

Like for the **`StackLayout`**, child views in the **`FlexLayout`** are positioned near each other, horizontally or vertically, but they are wrapped to a new row or column (depending on the orientation) if there is not enough space on screen. In its simplest form, you declare a **`FlexLayout`** as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    Padding="0,20,0,0"
    x:Class="Layouts.FlexLayoutExample">
   <FlexLayout Wrap="Wrap" JustifyContent="SpaceAround"
       Direction="Row">
    <Label Text="This is a sample label in a page"
     FlexLayout.AlignSelf="Center"/>
    <Button Text="Tap here to get things done"
       FlexLayout.AlignSelf="Center" x:Name="Button1"/>
   </FlexLayout>
</ContentPage>
```

**Tip: The `Padding` property has been assigned to the `ContentPage` object in order to add some distance between the top of the screen and the root visual element. This is usually required in iOS only, but for now, it applies to both platforms. You will learn how to handle properties by OS in the upcoming chapters.**

The **`FlexLayout`** exposes several exclusive properties:

- **`Direction`**: This property specifies whether child views should be arranged in a single row or column. It is of type **`FlexDirection`**, and supported values are **`Row`** (default), **`Column`**, **`RowReverse`**, and **`ColumnReverse`**. The last two will cause child views to be in the reverse order.
- **`Wrap`**: This property, of type **`FlexWrap`**, specifies whether child views must be wrapped to the next row or column (depending on the value of **`Direction`**) if there is not enough space in the first one. Allowed values are **`Wrap`** (wraps to the next row/column), **`NoWrap`** (forces the view content to stay in one row/column), and **`Reverse`** (wraps to the next row/column but in the reverse order).
- **`JustifyContent`**: This property, of type **`FlexJustify`**, specifies how child views should be organized in case there is extra space around

them. Possible values are start, center, end, **`SpaceAround`**, **`SpaceBetween`**, and **`SpaceEvenly`**. While start, center, and end are self-explanatory, the others deserve a more thorough explanation. With **`SpaceAround`**, the spacing between elements is set at one unit of space at the beginning and end, and at two units between them, so the elements and the space fill the row. With **`SpaceBetween`**, the space between views is equal between units, and there is no additional space at either the end of the row. With **`SpaceEvenly`**, the space between child elements is the same between each other and from the bounds of the parent view to the other elements.

Child views in the **`FlexLayout`** can be aligned via the **`FlexLayout.AlignSelf`** attached property, whose value can be **`Start`**, **`Center`**, **`End`**, and **`Stretch`**. *Figure 6.2* shows the result of the code example and demonstrates how child views wrapping works:

## The Grid

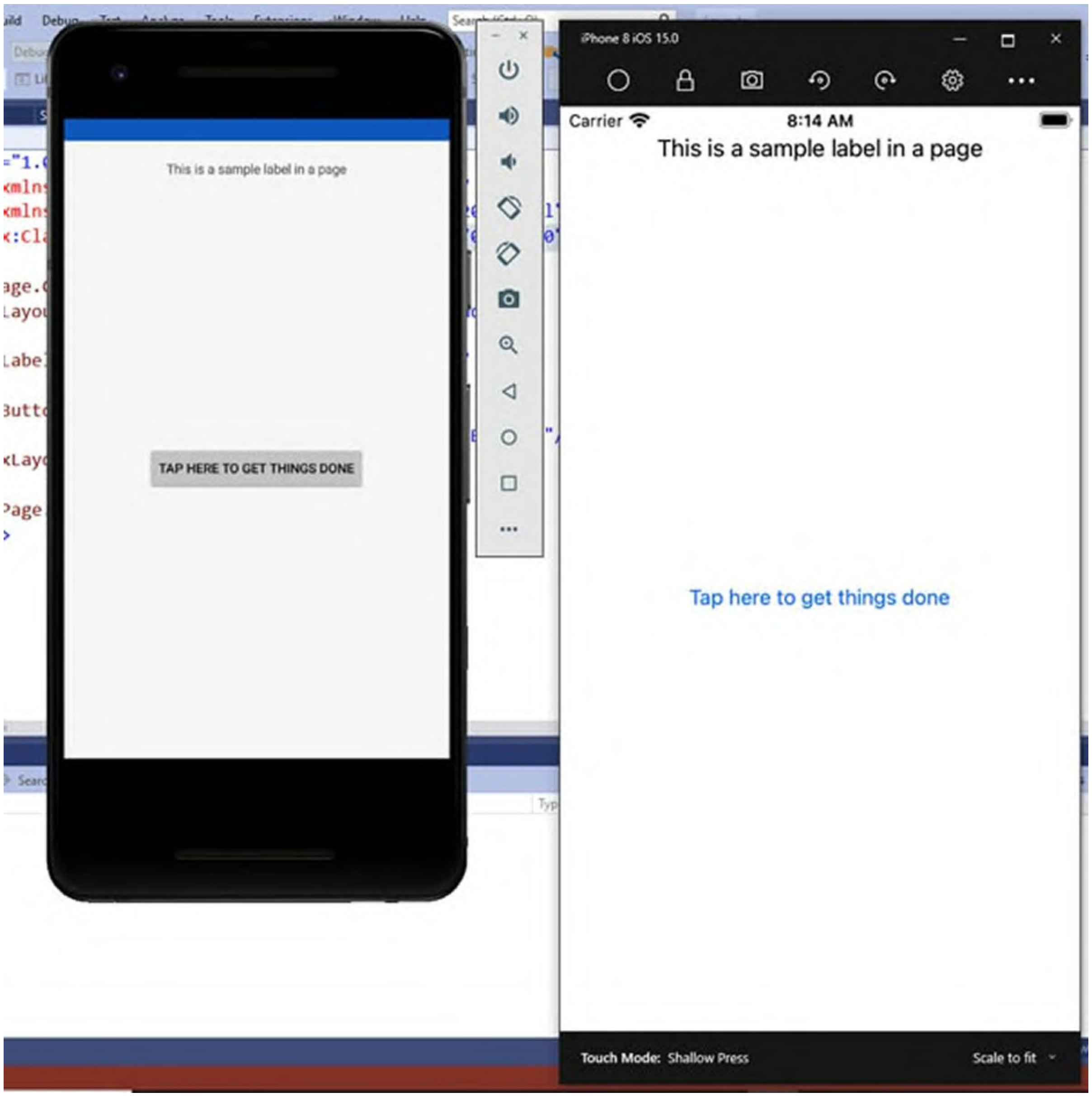

This is a sample label in a page
TAP HERE TO GET THINGS DONE
iPhone 8 iOS 15.0
Carrier
8:14 AM
This is a sample label in a page
Tap here to get things done
Touch Mode: Shallow Press
Scale to fit

***Figure 6.2:*** *Arranging views with the FlexLayout*

You can also try to assign the **`Wrap`** property with **`NoWrap`** to see how child views will be organized on the same row. Depending on the screen size of your device, views will overlap each other if not enough space is available. The **`FlexLayout`** is very versatile, because it is easy to use like a **`StackLayout`**, and it simplifies the way you implement child views for which you don't know the size in advance.

## The Grid

The **`Grid`** is the best layout in terms of rendering performance, and it allows you to organize child views within rows and columns, like in a virtual table. You can define only rows, only columns or both rows and columns to create virtual cells. Rows, columns, and cells can contain an individual view or another layout, which gives you great flexibility in designing a sophisticated user interface. The following code snippet shows how to define a **`Grid`** that is divided into two rows and two columns, creating four virtual cells:

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

**`RowDefinitions`** is a collection of **`RowDefinition`** objects, and **`ColumnDefinitions`** is a collection of **`ColumnDefinition`** objects, where each object represents a row or a column, respectively. You can also specify the **`Width`** or **`Height`** property to size rows and columns. If the **`Width`** and **`Height`** are not specified, both rows and columns will proportionally take the maximum space available, and this also makes it possible for rows and columns to be automatically resized if the parent view is resized. The previous code snippet shows how to create a virtual table with four cells. To add views to a **`Grid`**, you need to specify in which row and column the view needs to stay. This can be done by assigning the **`Grid.Row`** and **`Grid.Column`** properties, also referred to as attached properties, on the view declaration.

**As a general rule, attached properties make it possible to assign properties of another visual element from the current visual element.**

The index of rows and columns is zero-based, so 0 represents the first column from the left and the first row from the top. Rows, columns, and virtual cell can certainly contain nested layouts for more complex visual hierarchies. The following code demonstrates this, nesting grids:

```
<?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>
```

Note that the **`Grid.Row="0"`** and **`Grid.Column="0"`** values can be omitted for simplicity. The result of this code is shown in *Figure 6.3*:

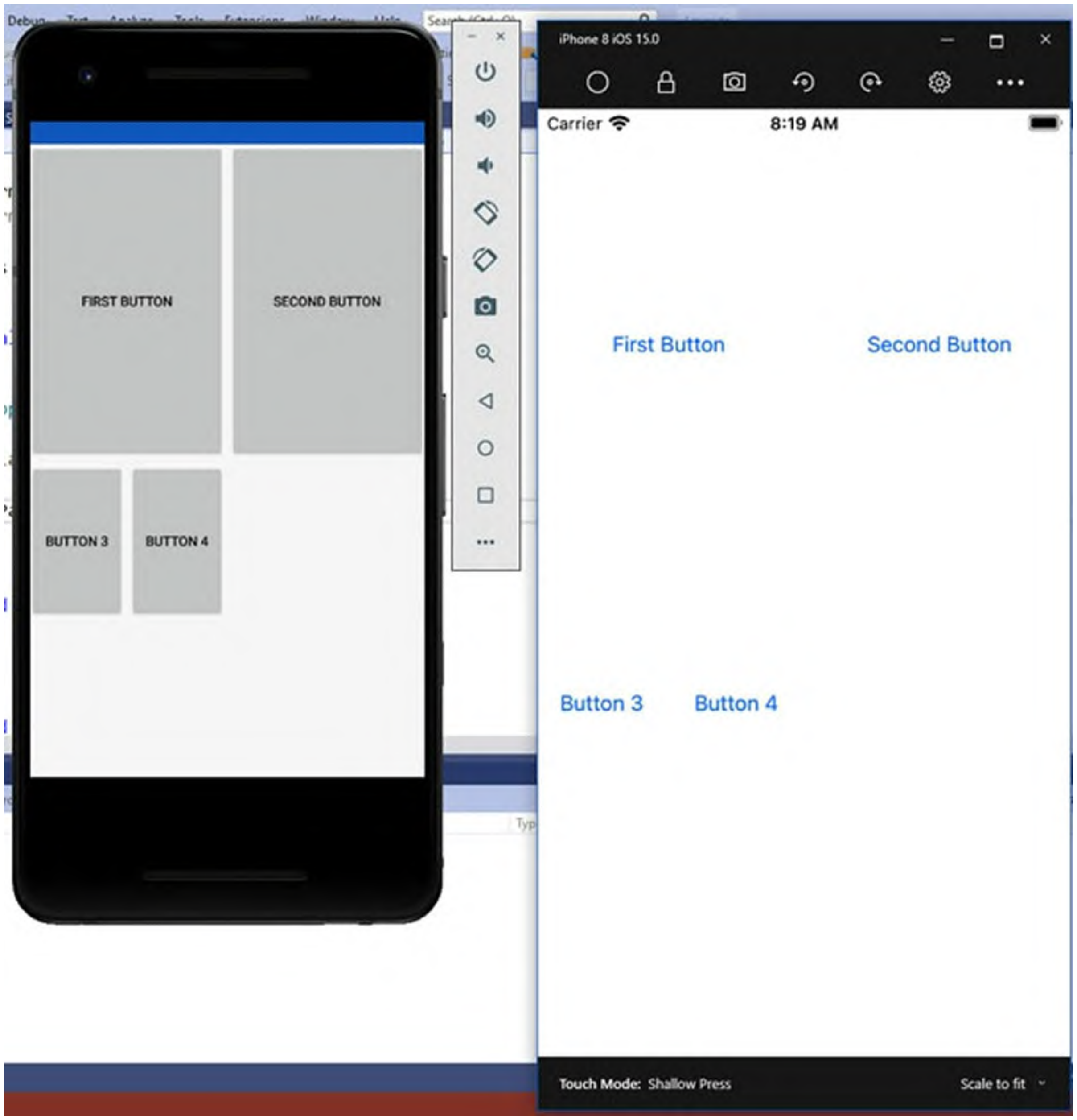


***Figure 6.3:*** *Tabular layout with a Grid*

## Spacing, proportions and spans for rows and columns

Rows and columns in a **`Grid`** should have dynamic height and width when possible. By default, rows and columns take all the available space, but you can additionally control the height of rows and the width of columns. The **`Height`** property of each **`RowDefinition`** and the **`Width`** property of each **`ColumnDefinition`** can be assigned with values from the **`GridUnitType`** enumeration so that you can manage their size, proportions and spacing.

Possible values are follows:

- **Auto:** Columns and rows are automatically sized to fit their content.
- **Star:** Columns and rows are proportionally sized to the remaining available space.
- **Absolute:** Columns and rows have fixed height and width.

In XAML, you use the `*` for star, and a numeric value for absolute, like in the following snippet:

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

You can also force visual elements in a cell to span across multiple rows or columns. This is accomplished via the **`Grid.RowSpan`** and **`Grid.ColumnSpan`** attached properties, assigned with the number of rows and columns that a view should take.

## The AbsoluteLayout

The **`AbsoluteLayout`** layout allows for the so-called absolute positioning, which means specifying the exact size and position of your visual elements based on their bounds. Let's look at an example for easier understanding:

```
<?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.AbsoluteLayoutExample" Padding="0,20,0,0">
  <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>
```

As you can see from the code, you specify the position of the child elements via the **`AbsoluteLayout.LayoutBounds`** and **`AbsoluteLayout.LayoutFlags`** attached properties. More specifically, the **`LayoutBounds`** property allows you to specify the position of the four bounds of your view with values of type double, separated by a comma. The **`LayoutFlags`** property accepts values from the **`AbsoluteLayoutFlags`** enumeration and allows you to deeper control the child view's position. The following is a list of possible values:

- **`All`**: All dimensions of child elements are proportional.
- **`HeightProportional`**: The height of the child element is proportional to the layout.
- **`WidthProportional`**: The width of the child element is proportional to the layout.
- **`None`**: No interpretation is done.
- **`SizeProportional`**: Combines **`WidthProportional`** and **`HeightProportional`**.
- **`XProportional`**: **`X`** property is proportional to the layout.
- **`YProportional`**: **`Y`** property is proportional to the layout.
- **`PositionProportional`**: Combines **`XProportional`** and **`YProportional`**.

*Figure 6.4* shows the result of the preceding code:

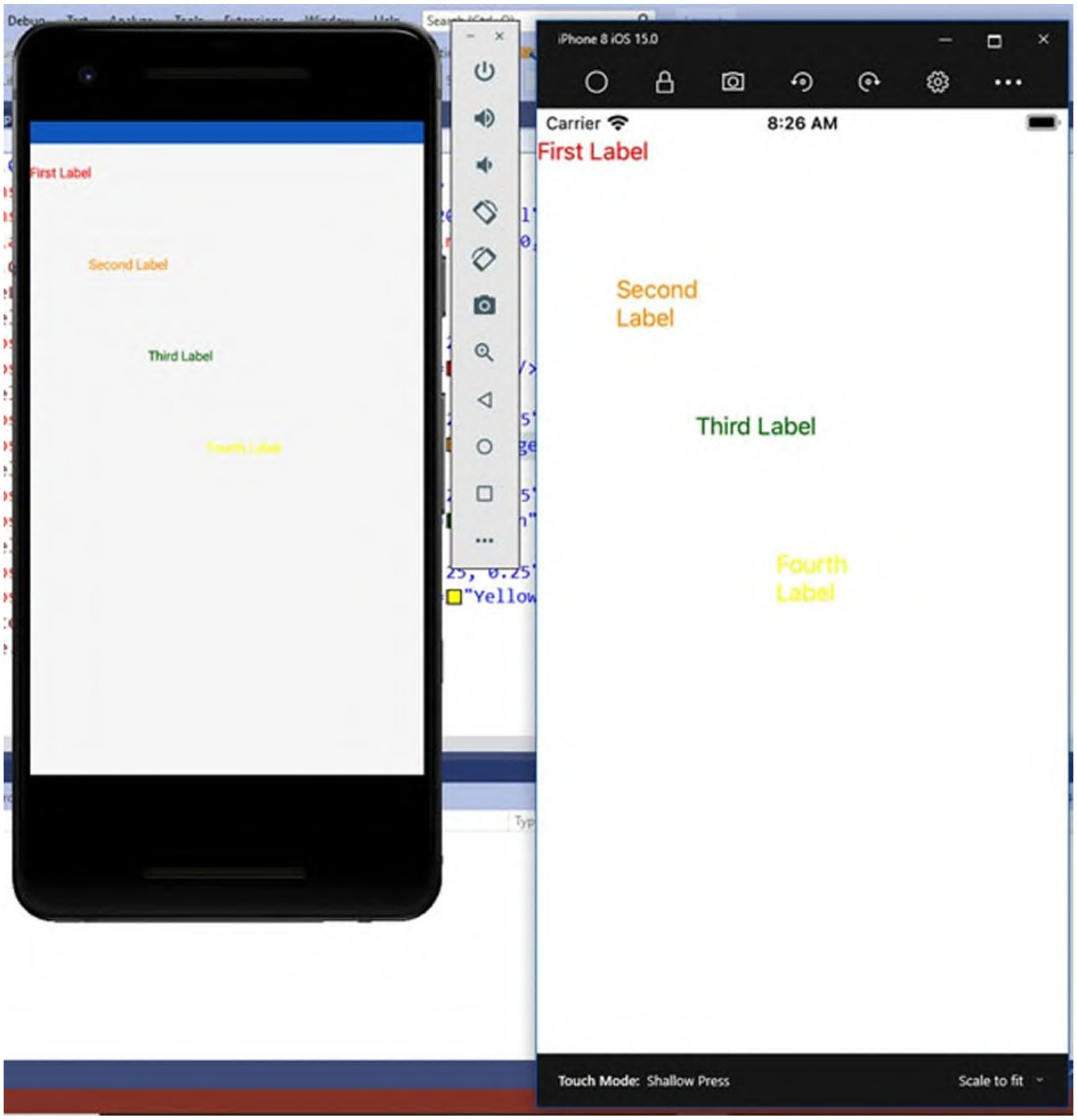


***Figure 6.4:*** *Fixed layouts with the AbsoluteLayout*

## The RelativeLayout

The **`RelativeLayout`** layout places child elements relative to each other or to the containing view. The position of child elements in a **`RelativeLayout`** is determined via so-called constraints, represented by the **`ConstraintExpression`** markup extension. This is responsible for specifying the size and position of a child view, considering the relationship of the child view to its parent or another named view.

**Markup extensions allow for obtaining a value that is not a primitive or a specific XAML type. The syntax requires an opening curly brace `{` to enter the markup extension scope, and a closing curly brace `}` to exit. Inside curly braces, you point to the desired type, which can be a resource or an object for databinding. You will see many more examples in *Chapter 9, Resources and Data Binding*.**

The **`RelativeLayout`** class exposes the **`XConstraint`** and **`YConstraint`** properties. The following code snippet demonstrates how to assign these properties with values coming from other visual elements, via attached properties. For the sake of simplicity, the example is based on the **`BoxView`** element, which basically draws a colored area:

```
<?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.RelativeLayoutExample">
  <ContentPage.Content>
    <RelativeLayout>
      <BoxView Color="Yellow" x:Name="YellowBoxView"
    RelativeLayout.YConstraint="{ConstraintExpression
    Type=RelativeToParent,
      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="Green"
    RelativeLayout.YConstraint="{ConstraintExpression
    Type=RelativeToView,
      ElementName=YellowBoxView,Property=Y,Factor=1,Constant=20}"
    RelativeLayout.XConstraint="{ConstraintExpression
    Type=RelativeToView,
      ElementName=YellowBoxView,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.Content>
</ContentPage>
```

*Figure 6.5* shows the result of this code:

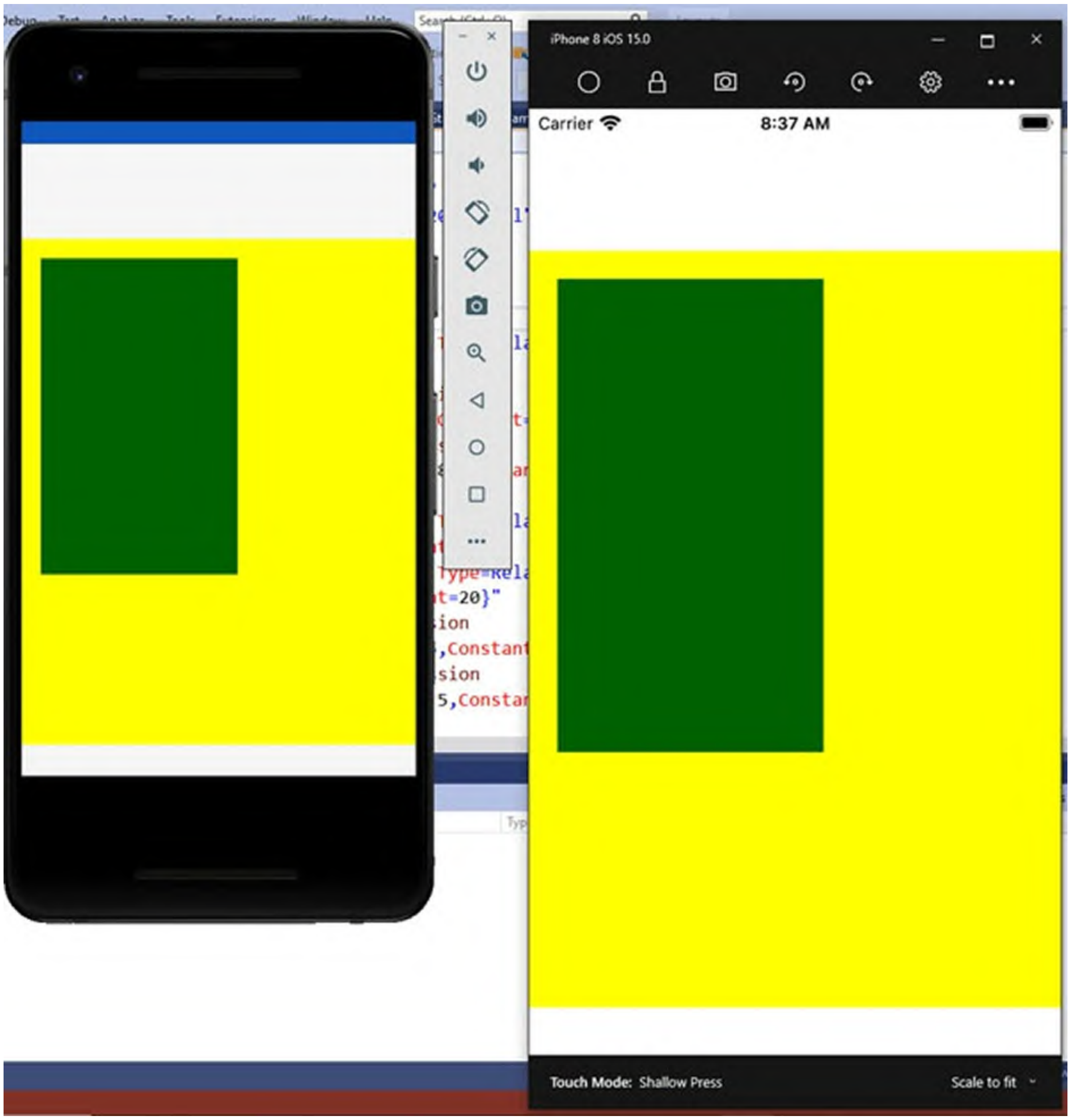


***Figure 6.5:*** *Relative positioning of views*

The official documentation recommends avoiding the **`RelativeLayout`** layout when possible due to poor rendering performance. If you really need to use it, try to at least avoid multiple **`RelativeLayout`** layouts in the same page.

## The ScrollView

Sometimes you might have a user interface that is made of a long list of visual elements that cannot fit on one screen, so they should be scrolled. To accomplish this, you include visual elements inside the **ScrollView** layout. It is very simple to use:

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

In short, a **ScrollView** adds scrolling capabilities to your visual hierarchy. In this example, because the height of the **BoxView** is quite big, it would not fit in one screen, so it will be possible to scroll the content. A common scenario with mobile apps is hiding the scroll bars, whose visibility can be changed by assigning the **HorizontalScrollbarVisibility** and **VerticalScrollbarVisibility** properties with the **Always**, **Never**, and **Default** values. So, you could assign **Never** to both properties. You can also control the scroll orientation via the **Orientation** property. Supported values are **Horizontal** and **Vertical**, so you can set the **ScrollView** to scroll only horizontally or only vertically. The action of scrolling is certainly not static, so providing a figure here that could not demonstrate the scrolling gesture would not make much sense. You can quickly run the companion code and see how it works on your own.

## Controlling the ScrollView programmatically

Sometimes, you might need to scroll a view in C# code; for example, you might need to focus on a particular visual element after the user has made a selection in the app. The **ScrollView** class exposes the **ScrollToAsync** method, which has two overloads. For a better understanding, consider the following two lines:

```
Scroll1.ScrollToAsync(0, 100, true);
Scroll1.ScrollToAsync(Label1, ScrollToPosition.Start, true);
```

In the first line, the method ensures that the content at 100 points from the top of the **ScrollView** is visible. In the second line, the method forces the **ScrollView** to move the specified visual element at the top of the view and ensures that the current position for the **ScrollView** is the same of the position of the specified view. The **ScrollToPosition** enumeration supports the following values:

- **Center**: Scrolls the child element to the center.

- **`End`**: Scrolls the child element to the end.
- **`MakeVisible`**: Makes the element visible within the view.
- **`Start`**: Scrolls the child element to the start.

**Avoid nesting `ScrollView` layouts, and avoid adding views with built-in scrolling (`CollectionView`, `WebView`, `ListView`, and so on) to a `ScrollView` because this might result in scrolling conflicts.**

## The Frame

The purpose of the **`Frame`** is to draw a colored border around the child visual element, and optionally, some shadow and circular corners. The following snippet provides an example:

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

Here’s a list and description of the most relevant properties:

- **`OutlineColor`**: Allows for specifying the color for the frame’s border.
- **`CornerRadius`**: Allows for drawing round corners around the Frame.
- **`HasShadow`**: When true, displays a drop shadow under the Frame.

*Figure 6.6* provides an example:

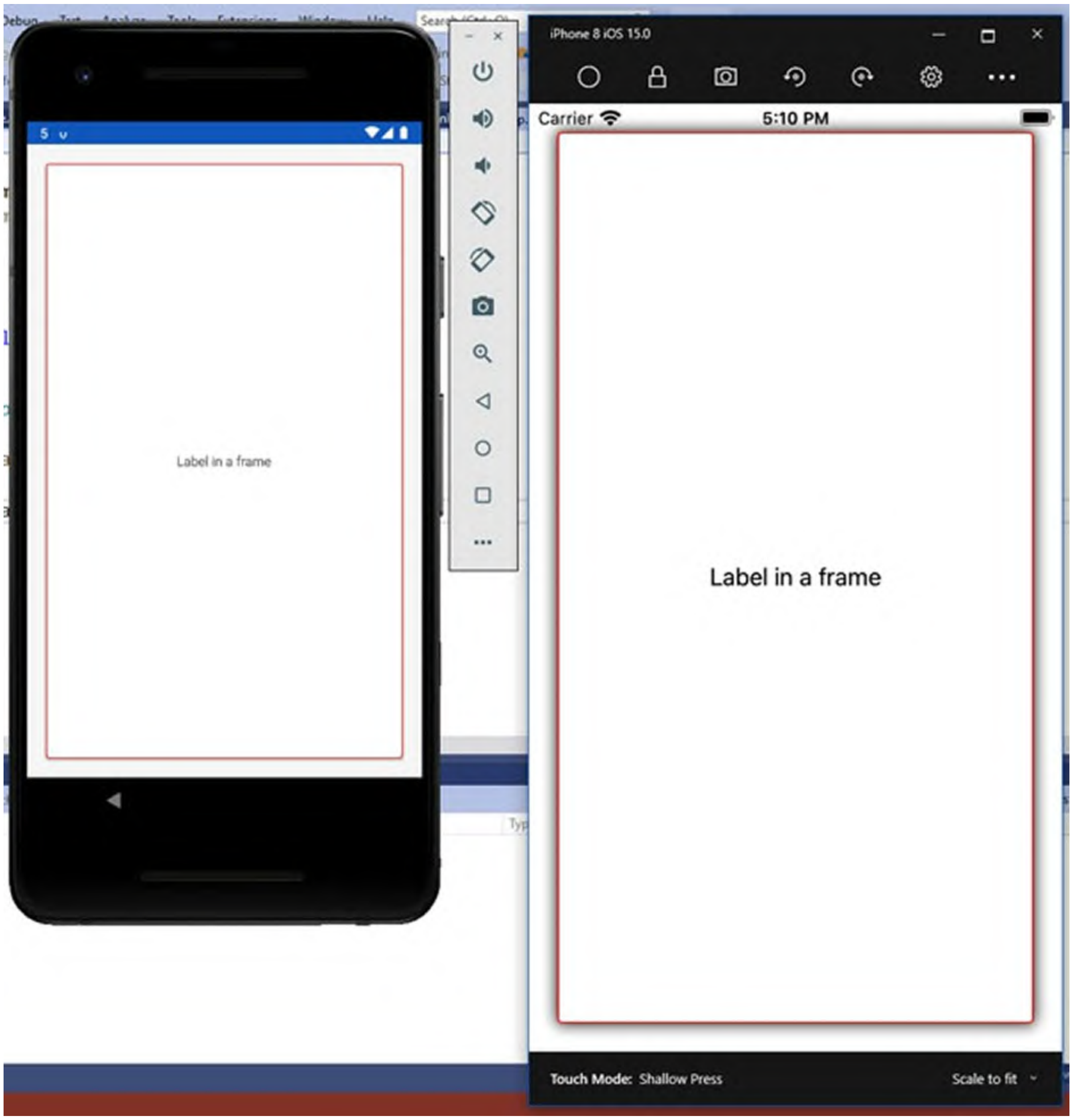


***Figure 6.6:*** *Surrounding views with a Frame*

## The ContentView

The **`ContentView`** layout is typically used to create custom, reusable controls because it makes it possible to combine multiple visual elements into a single view. To create a **`ContentView`**, in **`Solution Explorer`**, right-click on the shared project and then click on **`Add New Item`**. When the **`Add New Item`** dialog appears, click on the Xamarin.Forms node on the left, and then select the **`Content View item`**, as represented in *Figure 6.7*. Note that Visual

Studio also provides the **`Content View (C#)`** item template, but this generates a new class, where you will need to define your visual elements in C# rather than XAML:

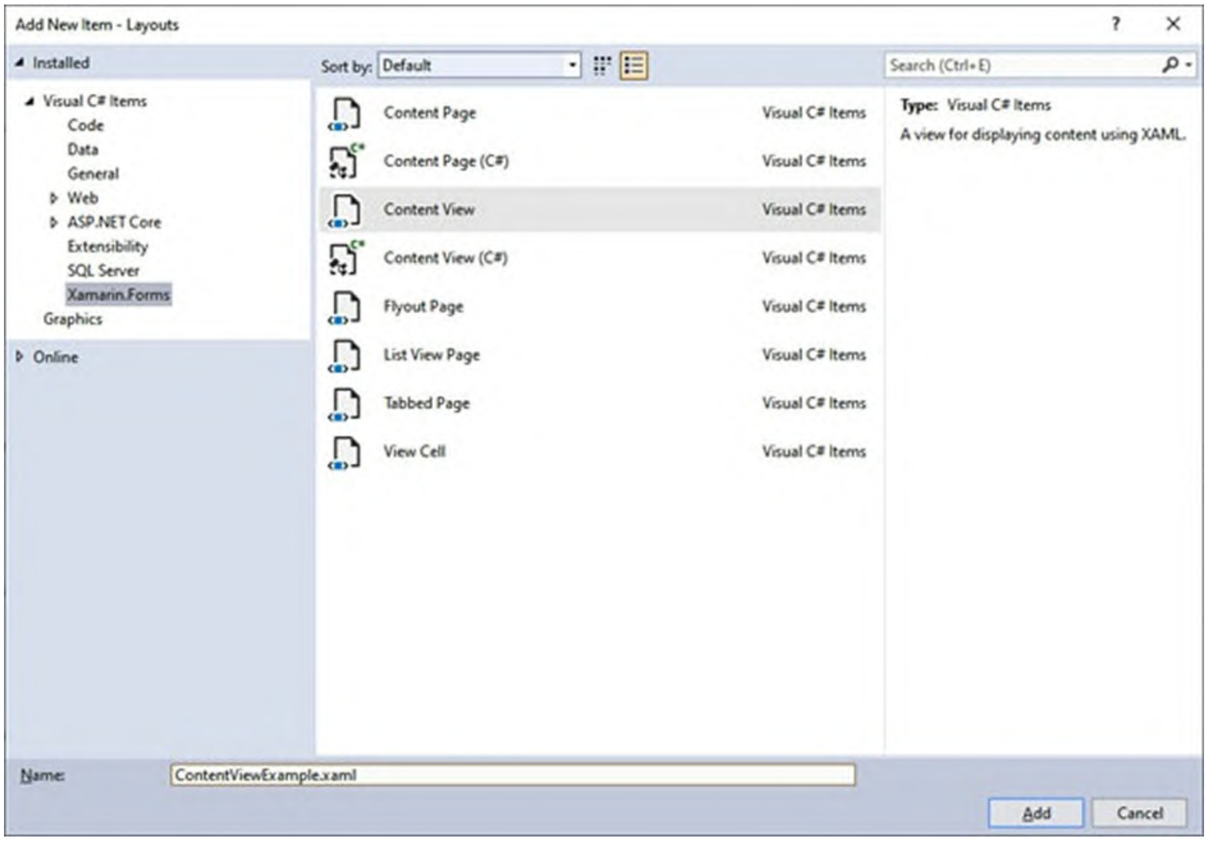

Add New Item - Layouts
Installed
Visual C# Items
Code
Data
General
Web
ASP.NET Core
Extensibility
SQL Server
Xamarin.Forms
Graphics
Online
Sort by: Default
Search (Ctrl+E)
Content Page — Visual C# Items
Content Page (C#) — Visual C# Items
Content View — Visual C# Items
Content View (C#) — Visual C# Items
Flyout Page — Visual C# Items
List View Page — Visual C# Items
Tabbed Page — Visual C# Items
View Cell — Visual C# Items
Type: Visual C# Items
A view for displaying content using XAML.
Name: ContentViewExample.xaml
Add
Cancel

***Figure 6.7:*** *Adding a new item to the project*

**The `Add New Item` dialog is also the tool you use to add any new file to the project, including pages and classes.**

At this point, the new item is added to the project. In the XAML editor, you can see how Visual Studio has added default visual elements, specifically, the **`ContentView`** root element, a **`StackLayout`** and a **`Label`**. Now, you can customize the **`ContentView`** by adding your own visual hierarchy, as shown in the following code. Then, the **`ContentView`** can be consumed like any other control or layout:

```
<?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="Layouts.ContentViewExample">
  <ContentView.Content>
   <StackLayout>
```

```
    <Label Text="Enter your email address:" />
    <Entry x:Name="EmailEntry"
    TextChanged="EmailEntry_TextChanged" />
   </StackLayout>
  </ContentView.Content>
</ContentView>
```

It is worth mentioning that a **`ContentView`** XAML file always has a code-behind C# file where you can manage the individual visual elements, such as handling events. If you look at the **`Entry`** definition, a view that allows for entering text, the **`TextChanged`** event is also being handled. In the C# code-behind, you could have the following code:

```
private void EmailEntry_TextChanged(object sender,
TextChangedEventArgs e)
{
  _emailAddress = e.NewTextValue;
}
private string _emailAddress;
public string EmailAddress
{
  get => _emailAddress;
}
```

There are some key points here that you can reuse for your daily work:

- A **`private`** field stores the string typed by the user.
- The content of the private field is exposed to the external world by a **`readonly`** property called **`EmailAddress`**.
- Having a public **`readonly`** property and a private field allows for consuming data from caller views without the possibility of direct access to the data.

This approach is best practice, but you can always give callers the option to edit data with a classic read/write property.

## Using a ContentView

Using a **`ContentView`** requires three steps:

1. Adding an XML namespace declaration in the XAML file using the **`ContentView`**.
2. Declaring an instance of the **`ContentView`**.
3. Assigning properties and handling the **`ContentView`** in code.

For a practical example, open the **`ContentViewExample.cs`** file and locate the namespace declaration. The namespace identifier is what you need to add to the XAML of the calling page, and in this case, it is **`Layouts`**. Now, open the **`MainPage.xaml`** file, assuming that you will use the custom view here. In the page declaration, add the following XML namespace:

```
xmlns:local="clr-namespace:Layouts"
```

The namespace identifier is up to you, but local is used quite often when referring to the first-level app namespace. The syntax for referring to the namespace that defines one or more custom controls is **`clr-namespace:`**. Then, you can simply add an instance of the view, as follows:

```
<local:ContentViewExample />
```

You can then assign properties, handle events, and do everything that you would do with any other Xamarin.Forms control. *Figure 6.8* shows how the sample **`ContentView`** looks:

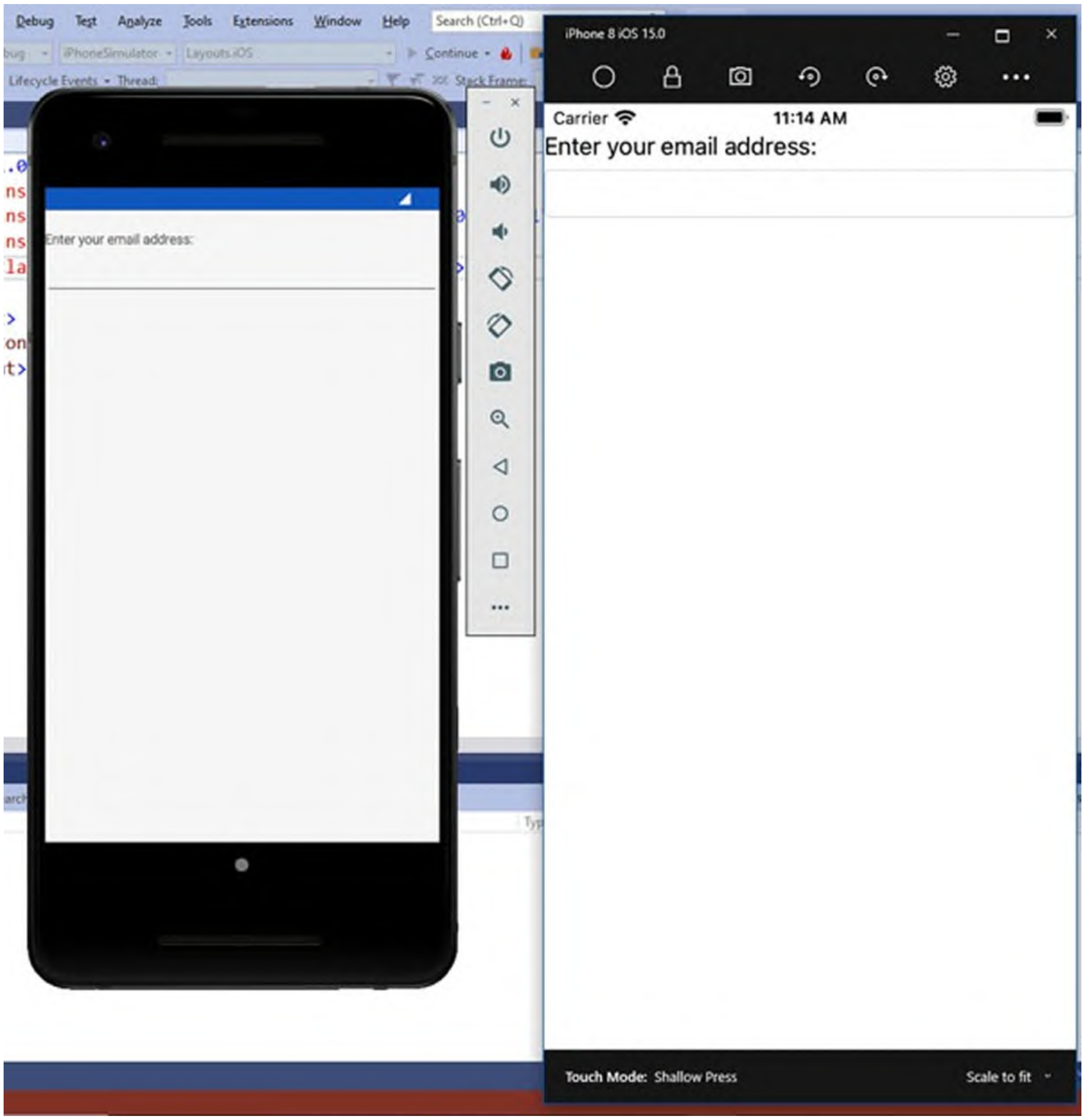


***Figure 6.8:*** *Adding a ContentView to the user interface*

If you assign a name to the view, you will be able to access its members from code; for example, the `EmailAddress` property defined previously.

## Styling the user interface with cascading style sheets

**Cascading style sheets** (**CSS**) allow for implementing different styles over the visual elements defined in a markup language like HTML. Xamarin.Forms supports a specific subset of CSS but not all the CSS elements and syntax. For this reason, this feature should be considered only as a complement to XAML, and it is recommended that you prefer XAML styles. Xamarin.Forms allows you to implement and consume CSS styles in your projects, two in XAML and one in C#.

## Defining CSS styles as a XAML resource

The first way you can implement and consume CSS styles in Xamarin.Forms is in XAML. You can add a **`StyleSheet`** object to the **`Resources`** collection of a page. The following code snippet demonstrates this:

```
<?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.CSSsample">
  <ContentPage.Resources>
   <ResourceDictionary>
    <StyleSheet>
     <![CDATA[
^contentpage {
background-color: lightblue; }
stacklayout {
margin: 20; }
]]>
    </StyleSheet>
   </ResourceDictionary>
  </ContentPage.Resources>
</ContentPage>
```

In this scenario, the CSS stylesheet definition is placed inside a CDATA section. You basically supply property values for each visual element in the form of key/value pairs. The syntax requires the visual element name and, enclosed within brackets, the property name followed by a colon, and the value followed by a semicolon, such as **`stacklayout { margin: 20; }`**. Note how the root element, **`contentpage`** in this case, must be preceded by the **`^`** symbol. You do not need to do anything else, as the style will be applied to all the visual elements specified in the CSS. *Figure 6.9* shows how the CSS style listed above changes the background color of the page into light blue, and how a margin is applied to the **`StackLayout`**:

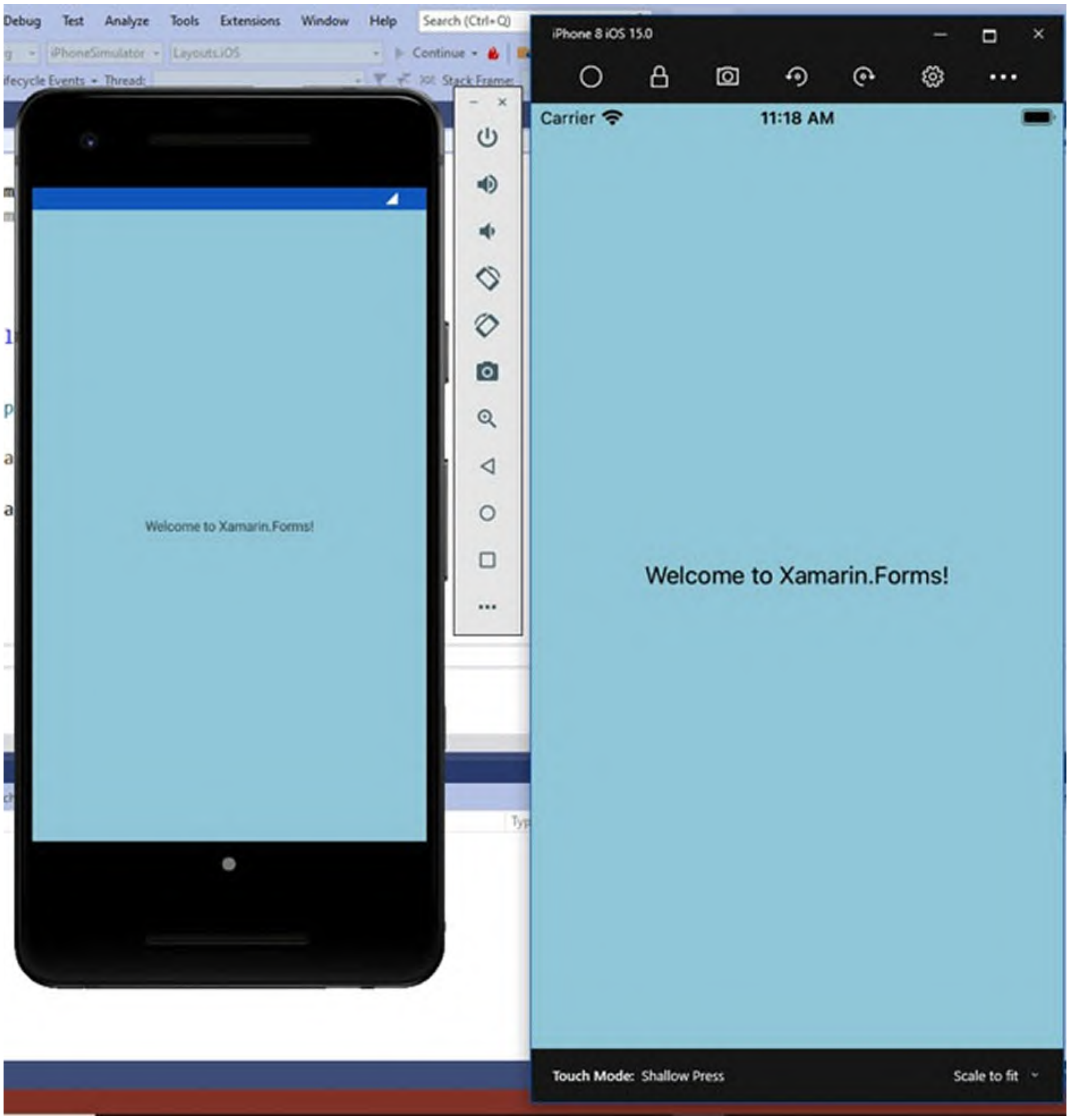

iPhone 8 iOS 15.0
Carrier
11:18 AM
Welcome to Xamarin.Forms!
Welcome to Xamarin.Forms!
Touch Mode: Shallow Press
Scale to fit

***Figure 6.9:*** *Styling the user interface with CSS*

**Names of visual elements inside the CSS definition must be lowercase.**

## Consuming CSS files in XAML

The second available option to consume a CSS style in XAML is from an existing `.css` file. First, you need to add your `.css` file to the `Xamarin.Forms` shared project and set its `BuildAction` property as `EmbeddedResource`. The `.css` file must follow the syntax supported by `Xamarin.Forms`. For example, you could copy and paste the content of the CDATA section, shown in the previous paragraph, into a new **.css** file. The next step is to add a **StyleSheet** object to a **ContentPage** object's resources and assign its **Source** property with the **.css** file name, like in the following example:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <StyleSheet Source="/mystyle.css"/>
  </ResourceDictionary>
</ContentPage.Resources>
```

Obviously, you can organize your **.css** files into subfolders; for example, the value for the **Source** property could be **/Assets/mystyle.css**. The companion code contains a sample **.css** file that is suitable for **Xamarin.Forms**.

## Creating and implementing CSS styles in C#

The last option for consuming CSS styles in **Xamarin.Forms** is C# code. You can create a CSS style from a string (through a **StringReader** object), or you can load an existing style from a **.css** file. However, in both cases, the key point is that you still need to add the style to a page's resources. The following code snippet demonstrates the first scenario, where a CSS style is created from a string and manually included in the **Resources** collection of the page:

```
using (var reader =
   new StringReader
     ("^contentpage { background-color: lightblue; }
     stacklayout { margin: 20; }"))
{
   // "this" represents a page
   // StyleSheet requires a using Xamarin.Forms.StyleSheets
   directive
   this.Resources.Add(StyleSheet.FromReader(reader));
}
```

For the second scenario, load the content of a CSS style from an existing file; an example is provided by the following code snippet:

```
var styleSheet = StyleSheet.FromResource
      ("Layouts.Assets.mystyle.css",
      IntrospectionExtensions.GetTypeInfo(typeof(CSSExample)).Ass
this.Resources.Add(styleSheet);
```

The second snippet is more complex, since the file is loaded via reflection (it requires using a **System.Reflection** directive in order to import the

**`IntrospectionExtensions`** object). Note how you provide the file name, including the project name (**`Layouts`**) and the subfolder (if any) name that contains the **`.css`** file.

**Reflection is one of the most complex yet interesting parts of developing for .NET. In short, it allows for investigating, reverse-engineering, creating, executing types and members in any .NET language. The official documentation for Reflection is available at https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection.**

## Conclusion

The user interface of a mobile app should always be able to automatically adapt to the screen size of different device form factors. In Xamarin.Forms, you accomplish this with layouts. The **`StackLayout`** arranges controls near one another, both horizontally and vertically. The **`FlexLayout`** does the same, but it is also capable of wrapping visual elements. The **`Grid`** arranges controls within rows and columns. The **`AbsoluteLayout`** allows for placing controls at an absolute position. The **`RelativeLayout`** arranges controls based on the size and position of other visual elements. The **`ScrollView`** allows for scrolling the content of visual elements that do not fit in a single page. The **`Frame`** can draw a border and shadow around a visual element. Finally, the **`ContentView`** allows for creating reusable controls. Xamarin.Forms also supports a subset of CSS stylesheets, which should only be considered as a complement to XAML, not a replacement. **`Layouts`** are containers for elements of the user interface, and the Xamarin.Forms codebase provides many, as you will learn in the next chapter.

## Points to remember

- The user interface of a mobile application should always be dynamic.
- In **`Xamarin.Forms`**, you use layouts to create dynamic user interfaces.
- You interact in C# with visual elements by assigning them a name via the **`x:Name`** tag.
- You can change the startup page in every **`Xamarin.Forms`** project by

assigning the **`MainPage`** property in the **`App.xaml.cs`** file.

- Avoid nesting controls with built-in scrolling into a **`ScrollView`**.
- You can add pages and content views to the shared project by right-clicking on the project name and then selecting **`Add New Item`**.

## Key terms

- **Layout**: A container for other visual elements.
- **View or Control**: Individual visual element with specific functionalities.

# CHAPTER 7
# Understanding Common Views

## Introduction

Views are the building blocks of the user interface in any mobile application, and they represent how the user performs activities in the app. For example, the date selector, a text box, and a calendar are all examples of views. **`Xamarin.Forms`** provide many views that are ready to use and that are described in this chapter. An important clarification is also necessary from the point of view of terminology. The word *view* generically refers to what you would call *control* in Microsoft terminology, *widget* in Android terminology, and view in Apple terminology. Though these could all be used interchangeably, for the sake of consistency with the official documentation, the view word will be used as the standard. All the concepts covered in the previous chapter will be very useful now because the views discussed in this chapter will be organized within layouts.

**Some of the views provided by Xamarin.Forms will be discussed in *Chapter 9, Resources and Data Binding*, rather than in this chapter. The reason is that such views work only with data-bound collections, so you first need to understand concepts behind databinding.**

## Structure

In this chapter, we will cover the following topics:

- Common properties
- Working with text
- User interaction with buttons
- Selecting dates and time
- Displaying HTML contents
- Selecting Boolean and numerical values

- Implementing search functionalities
- Handling long-running tasks
- Displaying images
- Adding interactivity to views
- Understanding visual states

## Objectives

After completing this chapter, you will be able to create basic mobile applications with Xamarin.Forms, display information to the user, implement the selection of different kind of values, and handle user interaction.

## Creating a sample project

This chapter comes with a companion **`Xamarin.Forms`** solution that you can open with Visual Studio to better follow the examples. If you wish to create a project on your own from scratch, you can follow these steps:

1. Create a new **`Xamarin.Forms`** solution called **`CommonViews`** for consistency with the sample solution.
2. Do not edit or remove the auto generated **`MainPage.xaml`** file; it will be used later.
3. For each layout discussed in the book, add a new item of type **`Content Page (XAML)`**. To accomplish this, right-click on the shared project name and then click on **`Add New Item`** in the context menu.
4. In the **`Add New Item`** dialog, click on the **`Xamarin.Forms`** node on the left and then select the **`Content Page (XAML)`** item template.
5. Assign to the new XAML file a name that matches the discussed view, for example, **`WorkingWithText.xaml`**, and click on **`Add`**.
6. For each page you add to the project, add an empty **`StackLayout`** to the **`ContentPage`** and assign its **`VerticalOptions`** property with **`CenterAndExpand`**. Unless where expressly specified, this will be the layout of choice for the code examples in the next pages.

## Common properties

All views derive from the **Xamarin.Forms.View** class, so they share a number of properties that are also of common use. The most relevant and most used properties in this book are described in *Table 7.1*.

| Property | Description |
|---|---|
| **HorizontalOptions** | Allows for specifying the horizontal alignment of a view. Supported values are the same as for layouts described in *Table 6.2* of *Chapter 6, Organizing the User Interface With Layouts*. |
| **VerticalOptions** | Allows for specifying the vertical alignment of a view. Supported values are the same as for layouts described in *Table 6.2* of *Chapter 6, Organizing the User Interface With Layouts*. |
| **HeightRequest** | Allows for specifying the height of a view. It is of type double. |
| **WidthRequest** | Allows for specifying the width of a view. It is of type double. |
| **IsVisible** | Of type bool, allows for specifying whether a view should be visible in the visual tree. The default is true. |
| **IsEnabled** | Of type bool, enables or disables a view. Disabling a view means it is visible, but the user cannot interact with it. The default is true. |
| **GestureRecognizers** | This property allows for adding interaction to views that do not support this natively. It is discussed in further detail in the *Adding Interactivity to Views* section later in this chapter. |

***Table 7.1:*** *Most relevant views' common properties*

It is important to have knowledge of these shared properties because they will be used often in the chapter.

## Working with text

Displaying and entering text is probably the most common operation within mobile apps. **Xamarin.Forms** provide three views about text: the Label, the Entry, and the Editor.

## Displaying text with the label

You use a **Label** view to display read-only text, such as headings, messages, and general contents. The following code snippet demonstrates how to declare a **Label** by assigning its most common properties:

```
<Label Text="Welcome to Xamarin for JobSeekers!"
TextColor="Blue"
```

```
    LineBreakMode="WordWrap"
    HorizontalTextAlignment="Center"
    VerticalTextAlignment="Center" FontSize="Medium"/>
```

The **`Text`** property is certainly the most important because it contains the text you want to display. You can set a color for the text via the **`TextColor`** property, and you can even assign the **`BackgroundColor`** property if you want the label to have a specific background color. The **`LineBreakMode`** property sets the way text should be truncated or wrapped to a new line. Supported values are described in *Table 7.2*:

| Value | Description |
|---|---|
| **`WordWrap`** | Wraps long text to a new line, ensuring that a full word is displayed before wrapping (default setting). |
| **`NoWrap`** | No wrapping or truncation is applied. |
| **`HeadTruncation`** | Truncates long text at its beginning. |
| **`TailTruncation`** | Truncates long text at its end. |
| **`MiddleTruncation`** | Truncates long text in the middle. |
| **`CharacterWrap`** | Wraps long text to a new line without ensuring a full word is displayed before wrapping, then only based on character boundaries. |

***Table 7.2:*** *Text truncation mode for the LineBreakMode enumeration*

**`HorizontalTextAlignment`** and **`VerticalTextAlignment`** set the alignment of text, not of the **`Label`**. Supported values are **`Start`**, **`Center`**, and **`End`**. You can also specify the size of the text by assigning the **`FontSize`** property. This is further explained in the managing fonts paragraph shortly. Additional useful properties are the following:

- **`CharacterSpacing`**: allows for specifying spacing between individual characters of the text.
- **`TextType`**: By default, a **`Label`** displays regular text, but you can assign this property with HTML and display HTML content.
- **`TextTransform`**: This property can convert the whole text into lowercase (**`LowerCase`**) and uppercase (**`UpperCase`**). No transformation is applied if this property is not expressly assigned.
- **`FlowDirection`**: Common to several views, this property makes it possible to support right-to-left text direction. Supported values are

**`LeftToRight`** (default), **`RightToLeft`**, and **`MatchParent`**. The latter inherits the direction from the parent view.

The **`Label`** view can also display formatted text, as you will see in the upcoming sections. *Figure 7.1* shows how the previous **`Label`** appears on both Android and iOS:

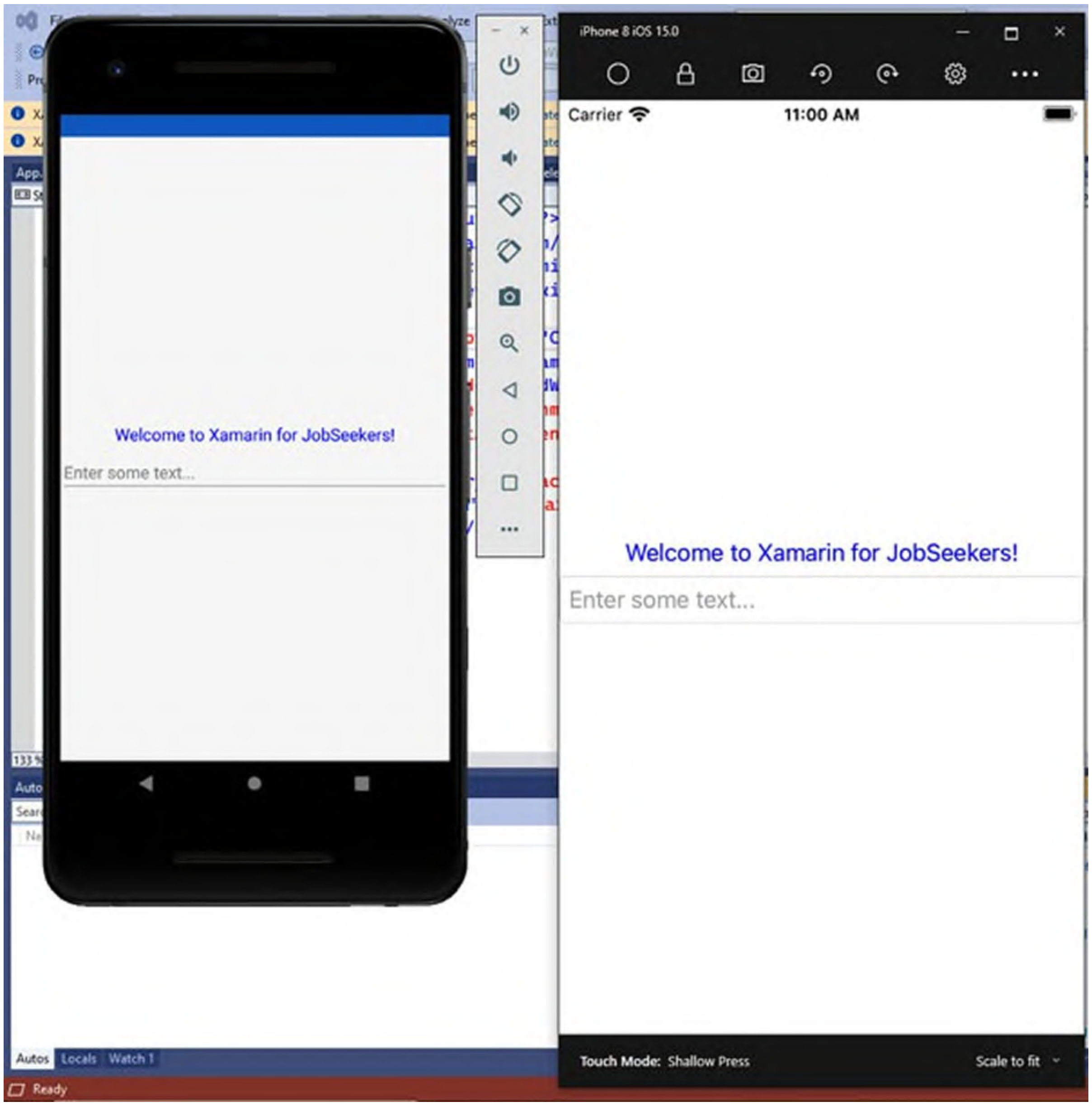


***Figure 7.1:*** *A Label and an Entry*

## Accepting user input with entry and editor

**`Xamarin.Forms`** provides two views to accept the user input: the Entry and the

Editor. The main difference is that the **`Entry`** allows for entering a single line of text, while the Editor allows for entering multiple lines. The following code snippet demonstrates how to declare an **`Entry`**:

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

*Table 7.3* summarizes the most relevant properties of the **`Entry`**.

**All the properties and events described for the Entry also apply to the Editor.**

| Property | Description |
|---|---|
| **`Text`** | Of type string, stores the text entered by the user. |
| **`Placeholder`** | Of type string, shows a description in the Entry that only appears when it is empty. |
| **`TextColor`** | Of type color, sets the color of the entered text. |
| **`Keyboard`** | Allows for displaying one of the built-in device keyboard types depending on the purpose of the view. Supported self-explanatory values are Chat, Default, Email, Numeric, Plain, Text, Telephone and URL. |
| **`ReturnType`** | Sets the text that is displayed on the Enter key of the device keyboard based on the following supported values: Default, Done, Search, Go, Next, Send. The text appears in the language set on the device. |
| **`IsSpellCheckEnabled`** | Of type bool, enables spell check over the input text if supported by the device culture. |
| **`IsTextPredictionEnabled`** | Of type bool, enables suggestions by the device keyboard. |

***Table 7.3:** Most relevant properties for Entry and Editor views*

The **`Entry`** also allows for setting properties like **`CharacterSpacing`**, **`HorizontalTextAlignment`**, and **`VerticalTextAlignment`** just like you would do with a **`Label`**. This is an interactive view, so it exposes events that you can handle to work with the entered text. The events you will more often handle are as follows:

- **`TextChanged`**: Fired at every keystroke.
- **`Completed`**: Fired when the system detects that the user is no longer typing.

You might have noticed that the definition of **`Entry`** earlier specifies a name for the view via the **`x:Name`** tag, and the reason is to make it possible to handle the view's events. The following code snippet demonstrates how to handle both events:

```
private void Entry1_Completed(object sender, EventArgs e)
{
  string enteredText = Entry1.Text;
}
private void Entry1_TextChanged(object sender,
TextChangedEventArgs e)
{
  string enteredText = e.NewTextValue;
  string previousText = e.OldTextValue;
}
```

With the **`Completed`** event, you retrieve the input via the **`Text`** property. With **`TextChanged`**, you can still do that, and you can also use the **`NewTextValue`** and **`OldTextValue`** properties of the **`TextChangedEventArgs`** class to understand which text was newly entered and compare it to the previous one. *Figure 7.1* shows the current Entry in action.

The Editor has the exact same behavior, properties, and events, but it allows for entering longer text over multiple lines.

### Entering passwords

The Entry exposes a useful property called **`IsPassword`**. When true, the user input is masked so that you can use the Entry to enter passwords. The result is still stored inside the **`Text`** property, and all the other behaviors and members remain unchanged.

## Applying and managing fonts

All the views that display text allow for customizing fonts, also known as typefaces. So, this not only applies to **`Label`**, **`Entry`** and **`Editor`** but to all the other views supporting text, like the button. For such views, the following properties allow you to manipulate fonts:

- **`FontFamily`**: Specifies the font to be used via its name.
- **`FontAttributes`**: Allows for formatting text as italic or bold via the same-named values, **`Italic`** and **`Bold`**.
- **`FontSize`**: Specifies the size of the text. This can be done either with

numeric values or with the so-called named size such as micro, small, medium, body (default), header, title, subtitle, caption, and large.

- **`TextDecoration`**: Allows for adding underline or strikethrough decorations via the same-named underline and strikethrough values.

When it comes to font size, it is recommended to use the named size over numeric values because using numeric values could have different results on devices with different OS and screen factor. A named size is consistent across platforms. The following line demonstrates how to display underlined text in bold and with header size:

```
<Label FontAttributes="Bold" FontSize="Header" Text="Bold
header"
   TextDecorations="Underline" />
```

## Implementing custom fonts

In the real world, it is extremely common to implement custom fonts. These can be either official fonts, such as from the Google Material design, or created by professional designers. In both cases, you need the **`.ttf`** font file. Once you have this, you can follow the given steps:

1. Add the **`.ttf`** files to the **`Assets`** folder of the Xamarin.Android project and to the **`Resources`** folder of the Xamarin.iOS project.
2. Set the **`Build Action`** property of the **`.tts`** file to **`AndroidResource`** for Xamarin.Android and to **`BundleResources`** for Xamarin.iOS.
3. In XAML, assign the new font to the **`FontFamily`** property of your views. The following code snippet provides an example with a **`Label`**:

```
<Label Text="Custom Font">
  <Label.FontFamily>
    <OnPlatform x:TypeArguments="x:String">
      <On Platform="iOS" Value="MyFont"/>
      <On Platform="Android" Value="MyFont#MyFont.ttf"/>
    </OnPlatform>
  </Label.FontFamily>
</Label>
```

This is the first time you see the **`OnPlatform`** tag. This will be thoroughly discussed later in the book, but for now, it is important to understand that this allows for differentiating the behavior of the same property on different platforms. The **`x:TypeArguments`** property specifies the .NET type for the property, which is string because this is the type for the **`FontFamily`** property.

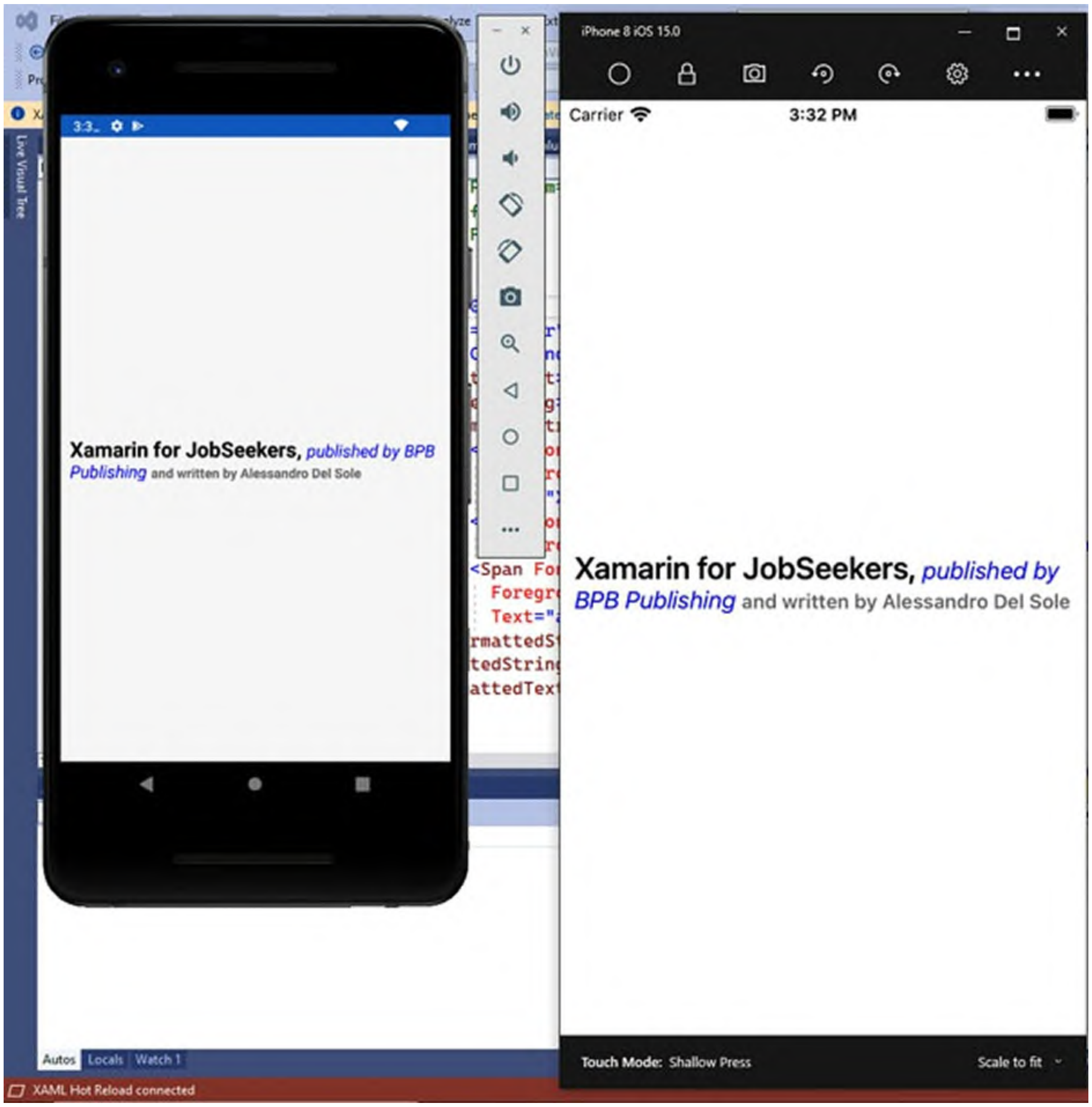
Next, for the iOS platform, the value of the property is assigned with the name of the font file without the **`.ttf`** extension. If you want to try with a real font, you can download some from Google (**https://fonts.google.com/**). All these fonts go well with Android and iOS devices, so you can have a more precise idea of how they work.

## Complex text formatting with FormattedString

The **`Label`** view provides the option to display text with more sophisticated string formatting. This is accomplished with the **`FormattedText`** property, of type **`FormattedString`**. Such a property is populated with a collection of **`Span`** objects, and each **`Span`** represents an area of the formatted string. The following code provides an example:

```
<Label Margin="10"
HorizontalOptions="Center"
VerticalOptions="CenterAndExpand">
  <Label.FormattedText>
    <FormattedString>
      <FormattedString.Spans>
        <Span FontSize="Large" FontAttributes="Bold"
          ForegroundColor="Black"
          Text="Xamarin for JobSeekers" />
        <Span FontSize="Medium" FontAttributes="Italic"
          ForegroundColor="Blue" Text="published by BPB Publishing"
          />
        <Span FontSize="Small" FontAttributes="Bold"
          ForegroundColor="Gray"
          Text="Written by Alessandro Del Sole" />
      </FormattedString.Spans>
    </FormattedString>
  </Label.FormattedText>
</Label>
```

The result of the preceding code is shown in *Figure 7.2*:



Xamarin for JobSeekers, *published by BPB Publishing* and written by Alessandro Del Sole

***Figure 7.2:*** *Implementing complex string formatting*

Span objects support text manipulation properties that you already saw when discussing the **`Label`** view, so you will be familiar with the **`FormattedString`** property.

## User interaction with buttons

The **`Button`** is probably the most used view for implementing user interaction. In a common flow of an app, a task starts to be performed after the user presses (or taps, in the mobile terminology) a button. The following code demonstrates how to invite the user to press a **`Button`** that is declared and designed via XAML:

```
<StackLayout Spacing="10" Margin="10"
VerticalOptions="CenterAndExpand">
  <Label Text="Tap the button"
   VerticalOptions="CenterAndExpand"
   HorizontalOptions="CenterAndExpand" />
  <Button x:Name="Button1" Text="Tap here!"
  BackgroundColor="DarkBlue"
    TextColor="White" BorderColor="LightBlue" BorderWidth="2"
    CornerRadius="4"
    Clicked="Button1_Clicked" />
</StackLayout>
```

*Table 7.4* summarizes the most important properties for a **`Button`**:

| **Property** | **Description** |
|---|---|
| **`Text`** | The text displayed in the button. |
| **`TextColor`** | The color for the text. |
| **`BorderColor`** | Adorns the button with a colored border. |
| **`BorderWidth`** | Specifies the thickness for the border around the button. |
| **`CornerRadius`** | Specifies the radius of the button’s corners. |
| **`Image`** | Allows for specifying an image that is placed close to the button text. |
| **`BackgroundColor`** | The background color for the button. |

***Table 7.4:*** *Most relevant Button properties*

The preceding XAML code will produce the result shown in *Figure 7.3*:

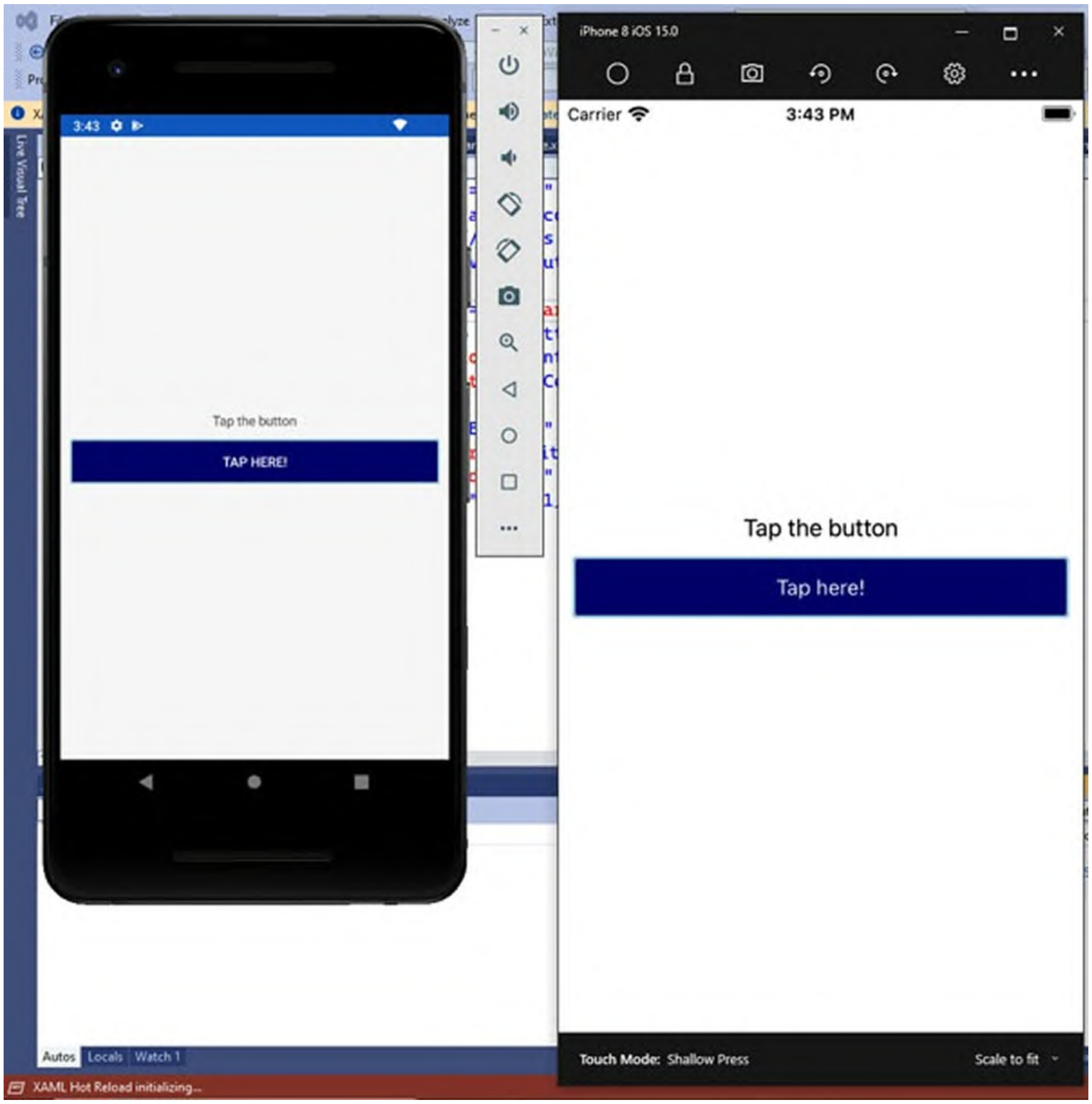


***Figure 7.3:*** *Implementing user interaction with buttons*

Note how the **`Button`** declaration includes a name for the view and a **`Clicked`** event that points to an event handler called **`Button1_Clicked`**. This is the place in the C# code-behind file where you start the action that is activated when the user taps the button. The event handler looks like this:

```
private void Button1_Clicked(object sender, EventArgs e)
{
  // Start an action here…
}
```

For example, you could move to another page in the app and download some data or anything that is relevant to do in the context of your application. Once you have full knowledge of what you can do with **Xamarin.Forms**, it will be easier for you to understand which actions best suit a specific context.

**Tip: Because the `Button` displays text, it also exposes properties like `FontFamily`, `FontSize`, `FontAttributes`, `TextTransform`. You use them exactly as you would use a `Label`.**

**Xamarin.Forms** also provide the **ImageButton** view, which works exactly like the Button but shows an image instead of text. The image you want to display is assigned to the Source property of the ImageButton, as follows:

```
<ImageButton Source="AnImage.png" x:Name="AnImageButton"
   Clicked="AnImageButton_Clicked"/>
```

All the other options to control the corner radius and border are still available, and you can still handle the **Clicked** event to raise an action. The **Source** property is of type **ImageSource**, and the way you assign images via an instance of this object is thoroughly discussed in the *Displaying images* section later in this chapter, so it is recommended that you keep that section as a reference.

## Selecting dates and time

In **Xamarin.Forms**, you can leverage each system's user interface to select dates and time via the **DatePicker** and **TimePicker** views, respectively. These are discussed in this section.

## Selecting dates with the DatePicker

The **DatePicker** is declared as follows:

```
<DatePicker x:Name="DatePicker1" MinimumDate="01/01/2021"
    MaximumDate="12/31/2021"
    DateSelected="DatePicker1_DateSelected"/>
```

You can specify an interval of valid dates with the **MinimumDate** and **MaximumDate** properties, both of type **DateTime**. Dates are represented with the **MM/DD/YYYY** format. The **Date** property contains the selected date and, if not specified in XAML, defaults to the system's current date. When the user selects a date, the **DateSelected** event is raised, and you can handle it as follows:

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

The **`DateChangedEventArgs`** class provides the **`NewDate`** property, which contains the newly selected date, and the **`OldDate`** property, which contains the previous date. *Figure 7.4* shows how the **`DatePicker`** looks on both platforms:

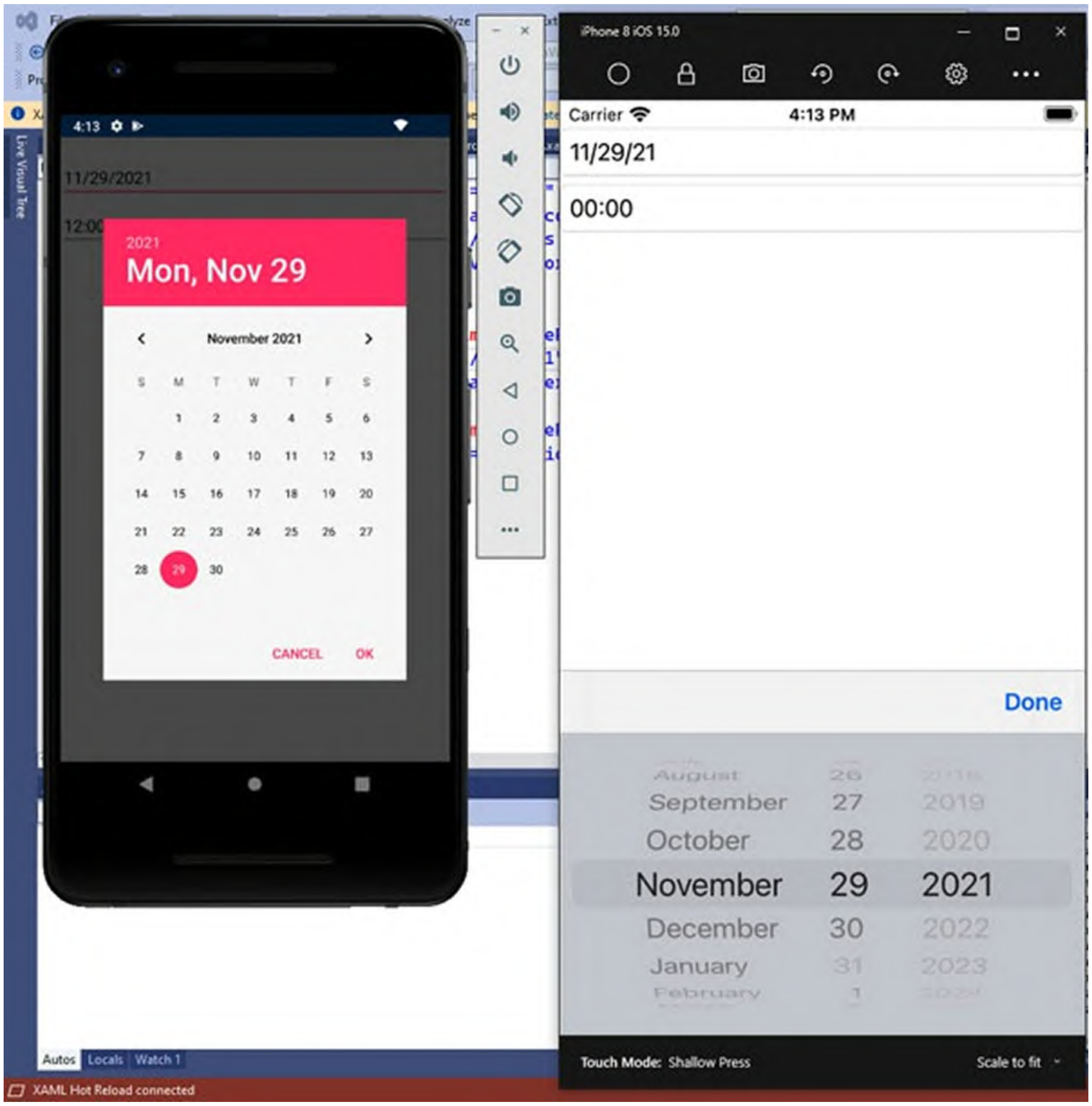


***Figure 7.4:*** *The DatePicker showing the system UI to pick up a date*

## Selecting a time with the TimePicker

Conceptually similar to the **`DatePicker`**, the **`TimePicker`** allows users to select a time of the day by leveraging the system user interface for this. A **`TimePicker`** can be declared as follows:

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

This view exposes a property called **`Time`**, of type **`TimeSpan`**, which represents the selected time. However, it does not expose an event that is raised when the user makes a selection, so the workaround is intercepting the **`PropertyChanged`** event, common to all views, checking for value changes over the **`Time`** property. This is accomplished by the following simple code:

```
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, and dependency properties will be discussed in *Chapter 9: Resources and Data Binding*. For now, you just need to know that **`PropertyName`** represents the name of the property that you want to intercept. *Figure 7.5* shows how the **`TimePicker`** appears on both Android and iOS:

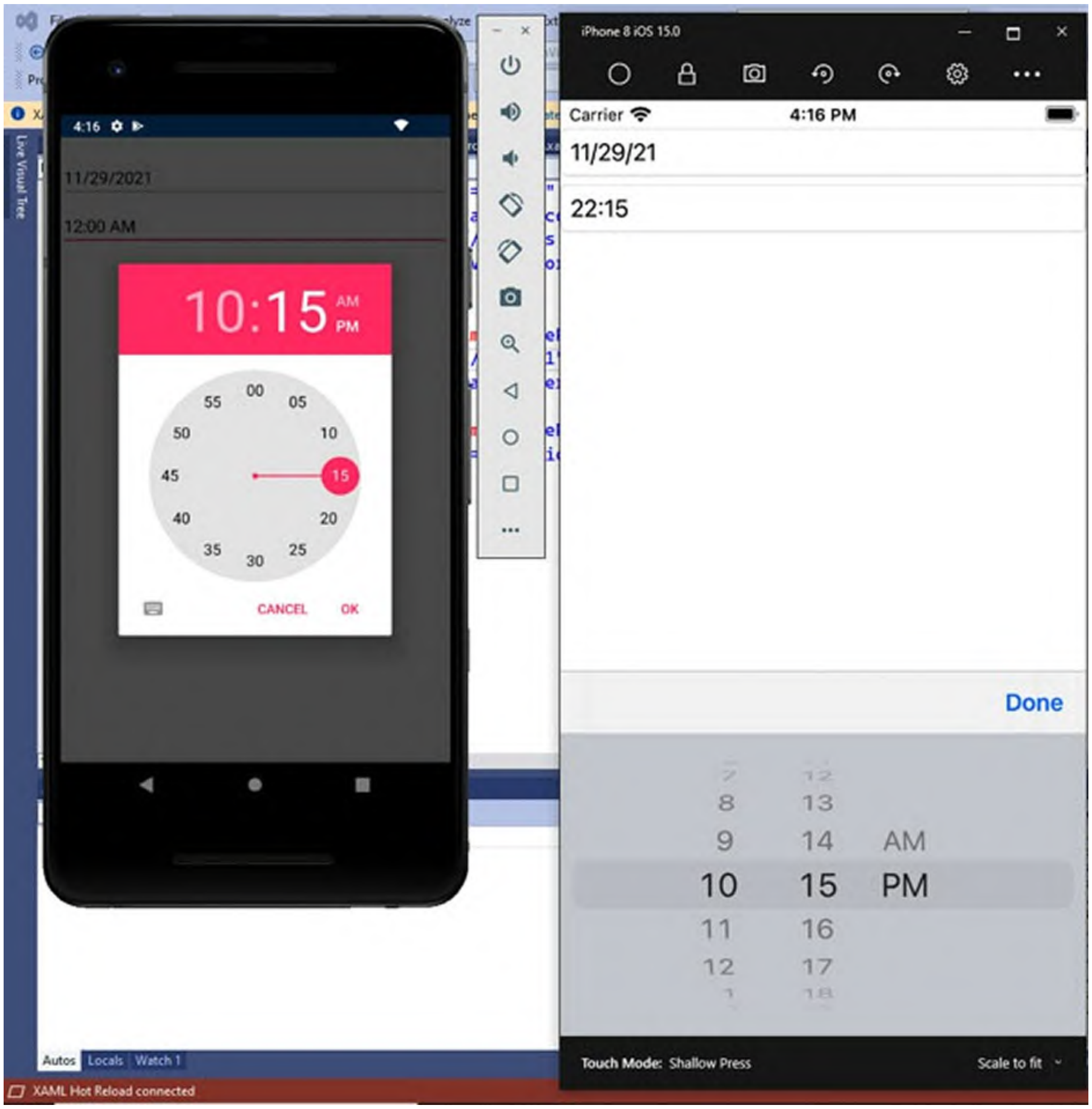


*Figure 7.5: Selecting a time of the day*

## Displaying HTML content

Sometimes, you might need to open a website or display an HTML static document directly inside the app. To do this, you can implement a **WebView** view. In its simplest form, you declare a **WebView** as follows:

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

The result of this code is shown in *Figure 7.6*, where you can see the **WebView** displaying a website:

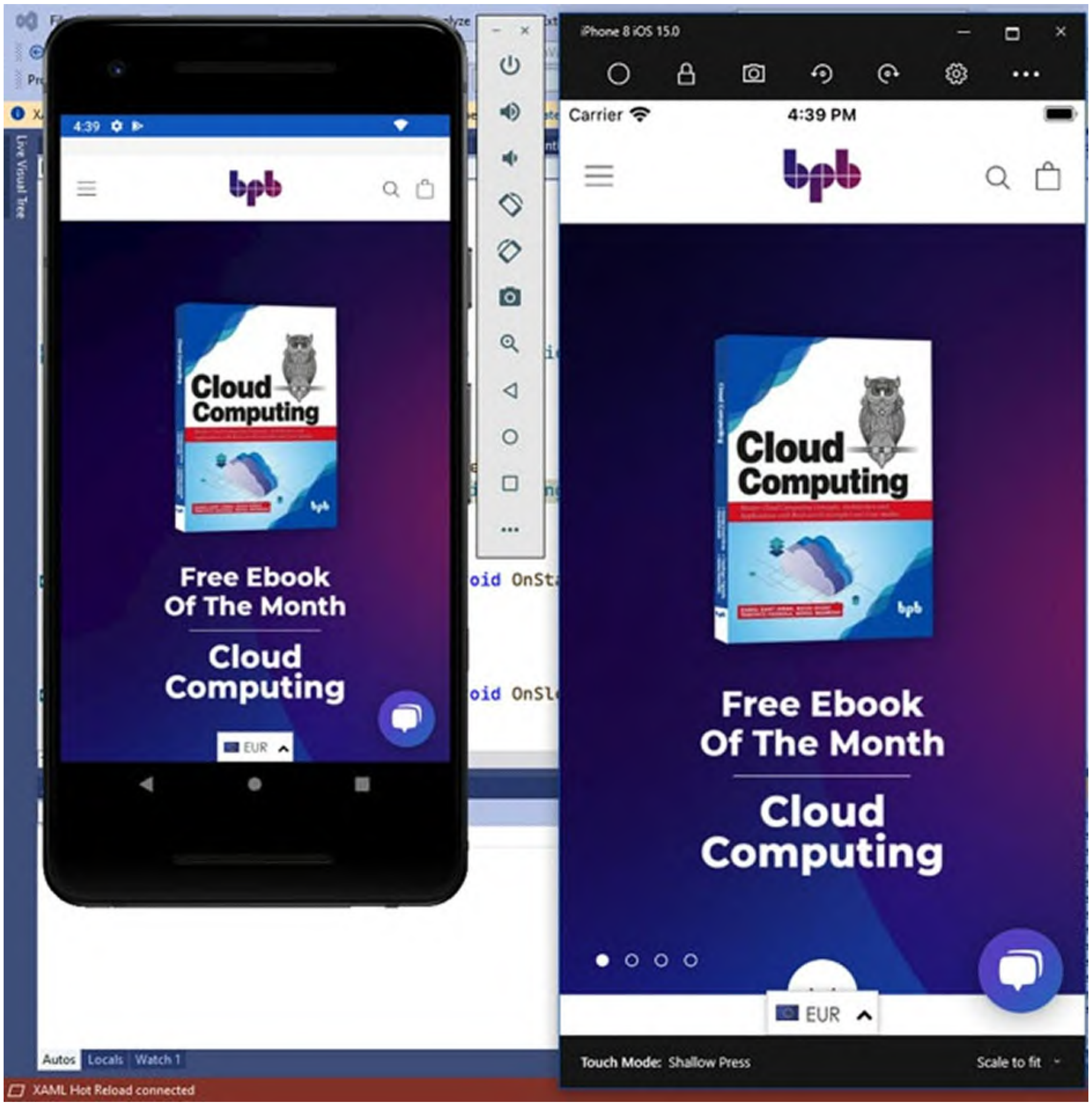


***Figure 7.6:*** *Opening a website with the WebView*

**Tip: On Android, if you use the `WebView` to open a website, remember to include the Internet permission in the app manifest.**

The **`WebView`** exposes the navigating event, which is raised when the navigation starts, and **`Navigated`**, which is fired when the navigation completes. Navigation can also be controlled programmatically via the **`GoBack`** and **`GoForward`** methods, which you invoke after checking that the **`CanGoBack`** and **`CanGoForward`** bool properties return true. The **`Source`** property of the **`WebView`** not only allows for specifying the source of the

HTML content, but it can also be assigned with different types. It is of type **WebViewSource**, an object that can receive URIs or strings containing HTML markup. For example, you can assign **Source** with static HTML content, as follows:

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

The best way to use a **WebView** is by including it inside a **Grid** for fully dynamic auto-sizing. If you add it inside a **StackLayout**, you will need to explicitly set the **WidthRequest** and **HeightRequest** properties.

## Selecting Boolean and numerical values

**Xamarin.Forms** provides several views to implement user input based on Boolean (true/false) and numerical values. These views are **Switch**, **CheckBox**, **Slider**, and **Stepper**.

## Turning options on and off with the switch

The **Switch** is a view that offers a toggled value and that you use to typically enable or disable options, and that in general you use to select true or false values. The value of the switch is set via the **IsToggled** property, whereas the **Toggled** event is fired when the user moves the **Switch** to a different position. It is important to mention that this view has no label, so you manually need to add one. The following example shows how to implement a **Switch**:

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

The user selection is stored inside an instance of the **ToggledEventArgs** object, which you retrieve by handling the **Toggled** event as follows:

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

*Figure 7.7* shows how the **Switch** looks:

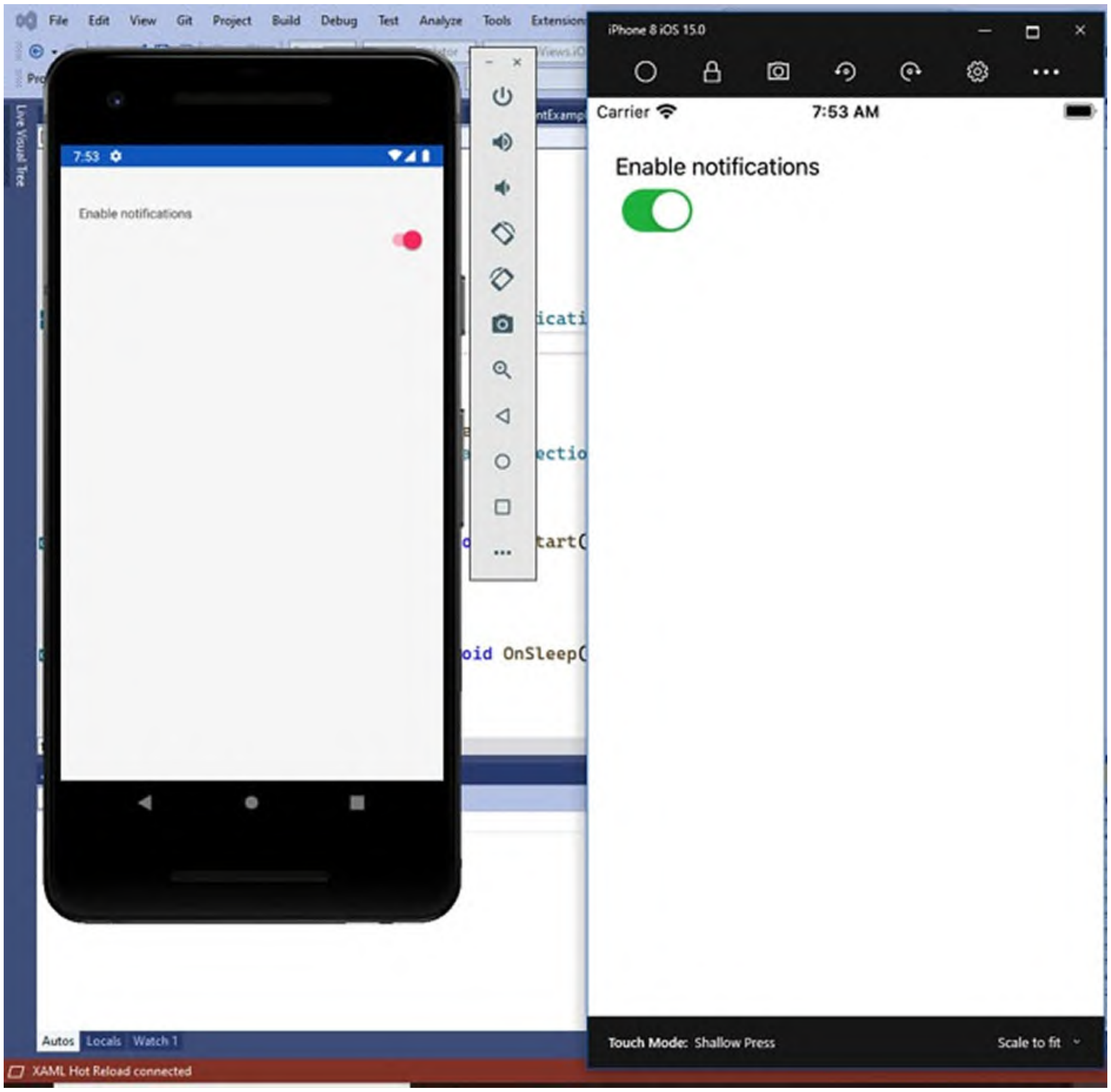


***Figure 7.7:*** *Turning options on and off with the Switch*

You can optionally supply a different color for the switch selector when it's turned on to the **OnColor** property.

## User choices with the CheckBox

The **CheckBox** is used to enable or disable user choices. Like the **Switch**, it does not have a built-in property to show text, so you need to do it yourself. The **IsChecked** property contains the true or false value of the selection, and the **CheckedChanged** event is raised when the user interacts with the view. The following code shows an example:

```
<StackLayout Orientation="Horizontal">
  <CheckBox x:Name="PrivacyBox" IsChecked="False"
     Color="Red" VerticalOptions="Center"
     CheckedChanged="PrivacyBox_CheckedChanged"
     Margin="5,0,0,0"/>
  <Label Text="I have read the privacy policy"
     VerticalOptions="Center"/>
</StackLayout>
```

Note how you can set the **`Color`** property with a specific color. The event handler for the **`CheckedChanged`** event will receive the information on the new value via the **`CheckedChangedEventArgs`** class, whose **`Value`** property contains the current value:

```
private void PrivacyBox_CheckedChanged(object sender,
CheckedChangedEventArgs e)
{
  bool newValue = e.Value;
}
```

*Figure 7.8* shows the result of the previous code:

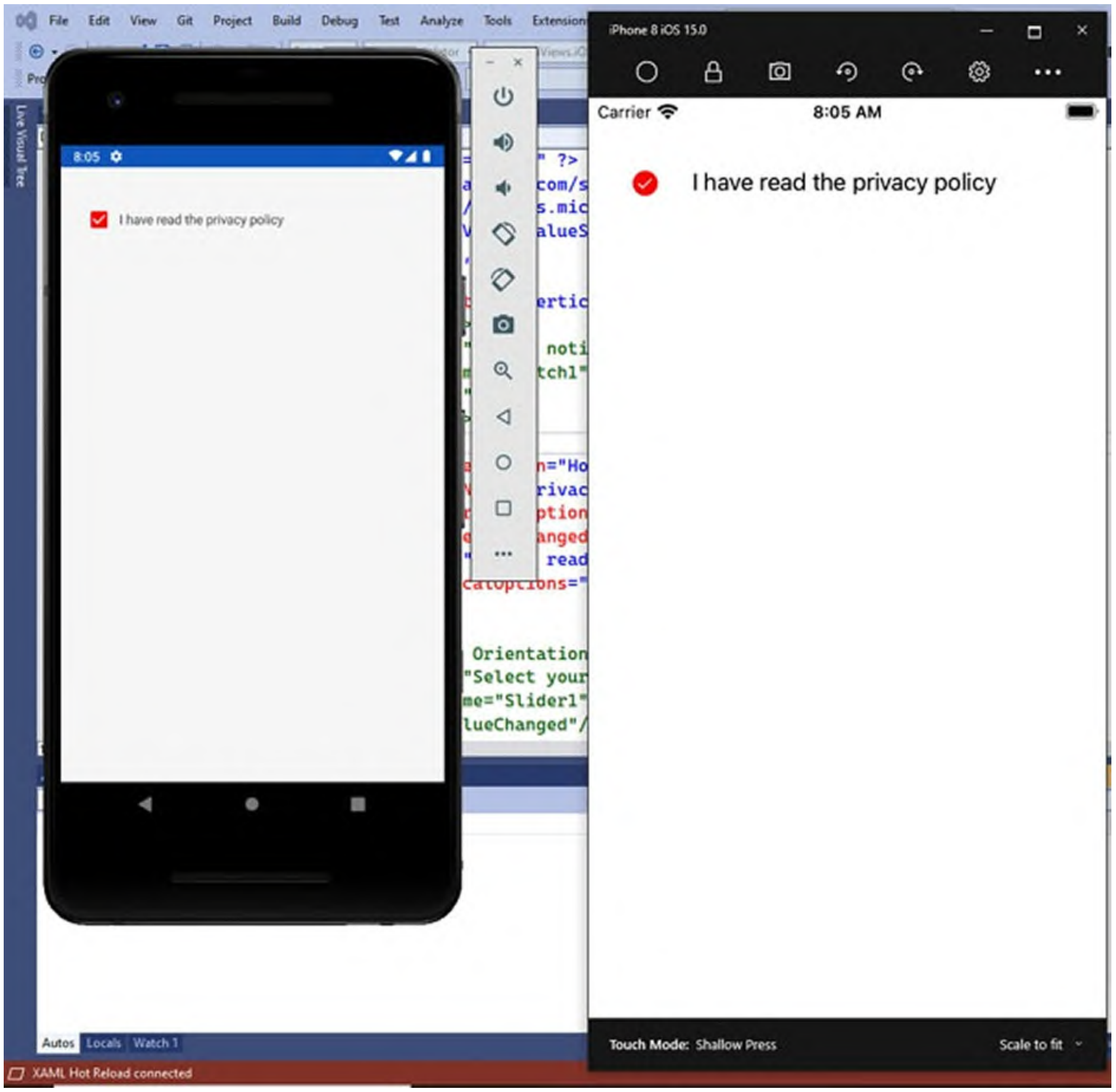

File Edit View Git Project Build Debug Test Analyze Tools Extensions
8:05
I have read the privacy policy
iPhone 8 iOS 15.0
Carrier
8:05 AM
I have read the privacy policy
Touch Mode: Shallow Press
Scale to fit
Autos Locals Watch 1
XAML Hot Reload connected

***Figure 7.8:*** *Implementing user choices with the CheckBox*

## Multiple choices with RadioButton

The `RadioButton` is a special view that allows for implementing mutually exclusive choices from a series. For better understanding, before looking at some code, consider *Figure 7.9* to see how the user can select one option from a list:

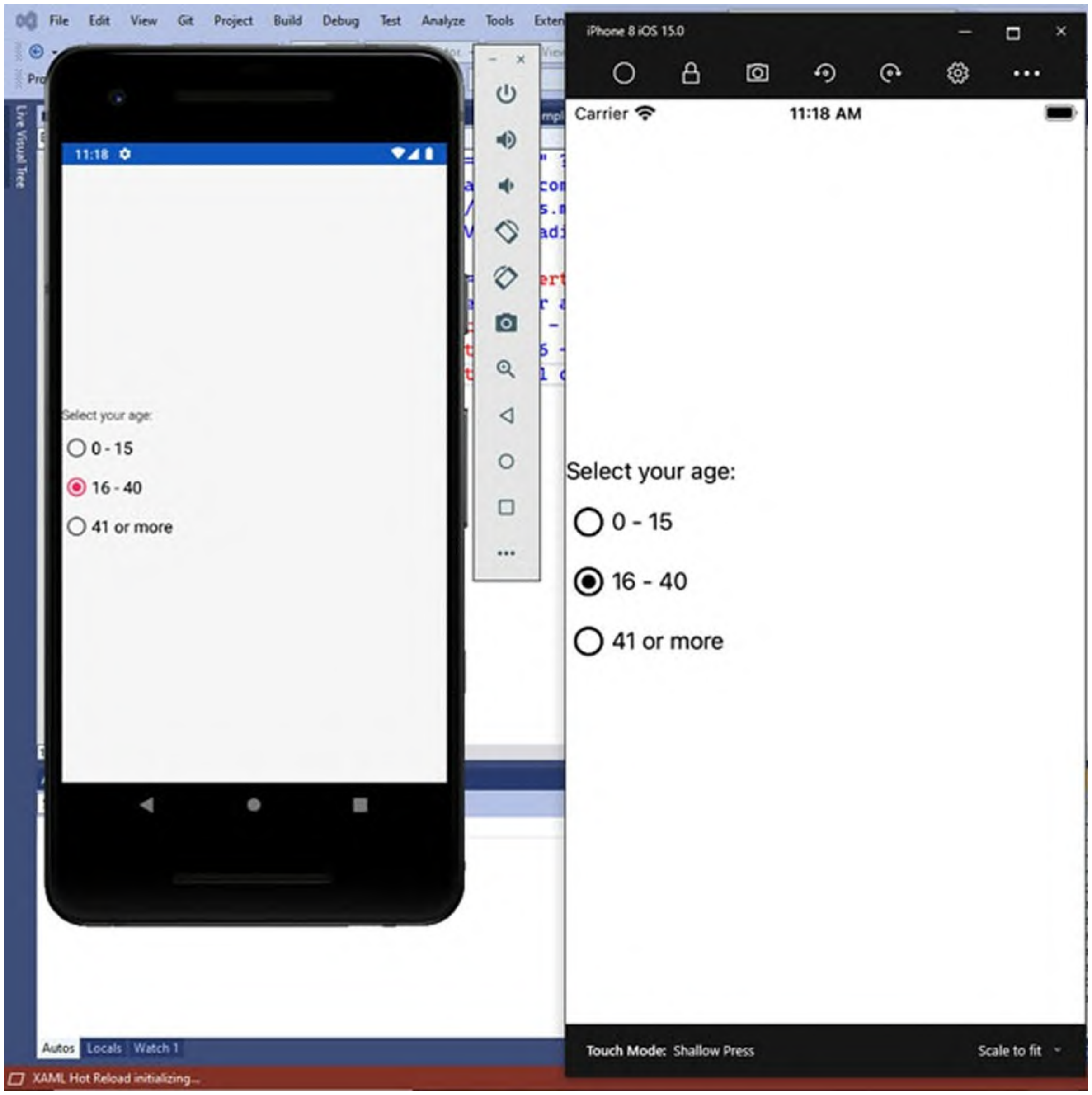


*Figure 7.9: Selecting an option from a list with the RadioButton*

When you select one option, all the others are disabled. You do not need to handle anything manually, and this is what mutually exclusive means. The following code implements the user interface in *Figure 7.9*:

```
<StackLayout Spacing="10" VerticalOptions="CenterAndExpand">
  <Label Text="Select your age:"/>
  <RadioButton Content="0 - 15" x:Name="FirstAgeRangeButton"
      CheckedChanged="FirstAgeRangeButton_CheckedChanged" />
  <RadioButton Content="16 - 40" x:Name="SecondAgeRangeButton"
      IsChecked="True"
      CheckedChanged="SecondAgeRangeButton_CheckedChanged"/>
  <RadioButton Content="41 or more" x:Name="ThirdAgeRangeButton"
```

```
      CheckedChanged="ThirdAgeRangeButton_CheckedChanged"/>
</StackLayout>
```

**RadioButton** views automatically work in groups. You can manually select one by assigning the **IsChecked** property with true, and this is also the property you use to get the status of a choice. The **Content** property is of type object and can contain either a string, like in the current example, or a view. For example, you could assign the **Content** property with an image:

```
<RadioButton Value="Option1">
  <RadioButton.Content>
    <Image Source="Option1Image.jpg" />
  </RadioButton.Content>
</RadioButton>
```

When the **Content** is assigned with a view, it is convenient to also assign a string value to the **RadioButton** via its **Value** property. Though not mandatory, this makes it possible to work against the value of a **RadioButton** rather than its display content. When a **RadioButton** is selected, the **CheckedChanged** event is fired. The event handler method stub looks like the following:

```
private void FirstAgeRangeButton_CheckedChanged(object sender,
    CheckedChangedEventArgs e)
{
  // if e.Value is true, the option was selected. Apply your
  logic here
}
```

You check the value of the **Value** property from the **CheckedChangedEventArgs** class to get the status of the **RadioButton**. If true, this was selected; otherwise, the selection was removed. If the **Content** property is assigned with a string, you can apply all the properties that you learned with the **Label**, such as **TextColor**, **TextTransform**, **FontFamily**, **FontSize**, and **FontAttributes**. On iOS, the **BorderColor** property also allows for drawing a colored border around the whole view.

## Implementing multiple groups

**RadioButton** views work in groups. When you add some, a group is automatically added and handled by the runtime. However, you might want to have multiple groups. For example, one group about the user age, another group to select a job title from a list, and so on. You have two options to specify a group name: the first option is adding a **RadioButtonGroup.GroupName** attached property to the parent layout of your

**RadioButton** views; the second option is assigning the **GroupName** property on each **RadioButton**. The following code demonstrates how to implement two different groups of options (for the sake of simplicity, names and event handlers have not been specified):

```
<StackLayout Spacing="10" VerticalOptions="CenterAndExpand">
  <StackLayout RadioButtonGroup.GroupName="Age">
    <Label Text="Select your age:"/>
    <RadioButton Content="0 - 15" />
    <RadioButton Content="16 - 40" />
    <RadioButton Content="41 or more" />
  </StackLayout>
  <StackLayout RadioButtonGroup.GroupName="Job">
    <Label Text="Select your job:"/>
    <RadioButton Content="Entrepreneur" />
    <RadioButton Content="Consultant" />
    <RadioButton Content="Employee" />
  </StackLayout>
</StackLayout>
```

If you try to run this code, you will see how the views in the first group do not interfere with the views in the second group, still being mutually exclusive between views in the same group.

## Value selection with the Slider

The **Slider** allows for selecting a value on a range delimited by a minimum and a maximum, represented by the **Value**, **Minimum**, and **Maximum** properties, of type double. The following code shows how to implement a **Slider**:

```
<StackLayout VerticalOptions="CenterAndExpand"
     HorizontalOptions="FillAndExpand" Spacing="10">
  <Label Text="Select a value: “/>
  <Slider x:Name="Slider1" Maximum="200" Minimum="10" Value="50"
     ValueChanged="Slider1_ValueChanged" ThumbColor="Blue"
     MinimumTrackColor="Red" MaximumTrackColor="Green"/>
  <Label Text="You selected: “ x:Name="ValueLabel" />
</StackLayout>
```

**Tip: With the Slider, the order of properties matters. In fact, you must specify the Maximum before the Minimum; otherwise, an exception will be thrown.**

Custom colors are totally optional; however, you can specify a color for the selector (**ThumbColor**), a color for the slider on the minimum side

(**`MinimumTrackColor`**), and a color for the slider on the maximum side (**`MaximumTrackColor`**). When the selector moves to a different value, the **`ValueChanged`** event is fired, and the new value is stored inside the **`NewValue`** property from the **`ValueChangedEventArgs`** object. You can retrieve it in the event handler, as follows:

```
private void Slider1_ValueChanged(object sender,
ValueChangedEventArgs e)
{
  double value = e.NewValue;
  ValueLabel.Text = $"You selected: {value}";
}
```

*Figure 7.10* shows the result of the preceding code:

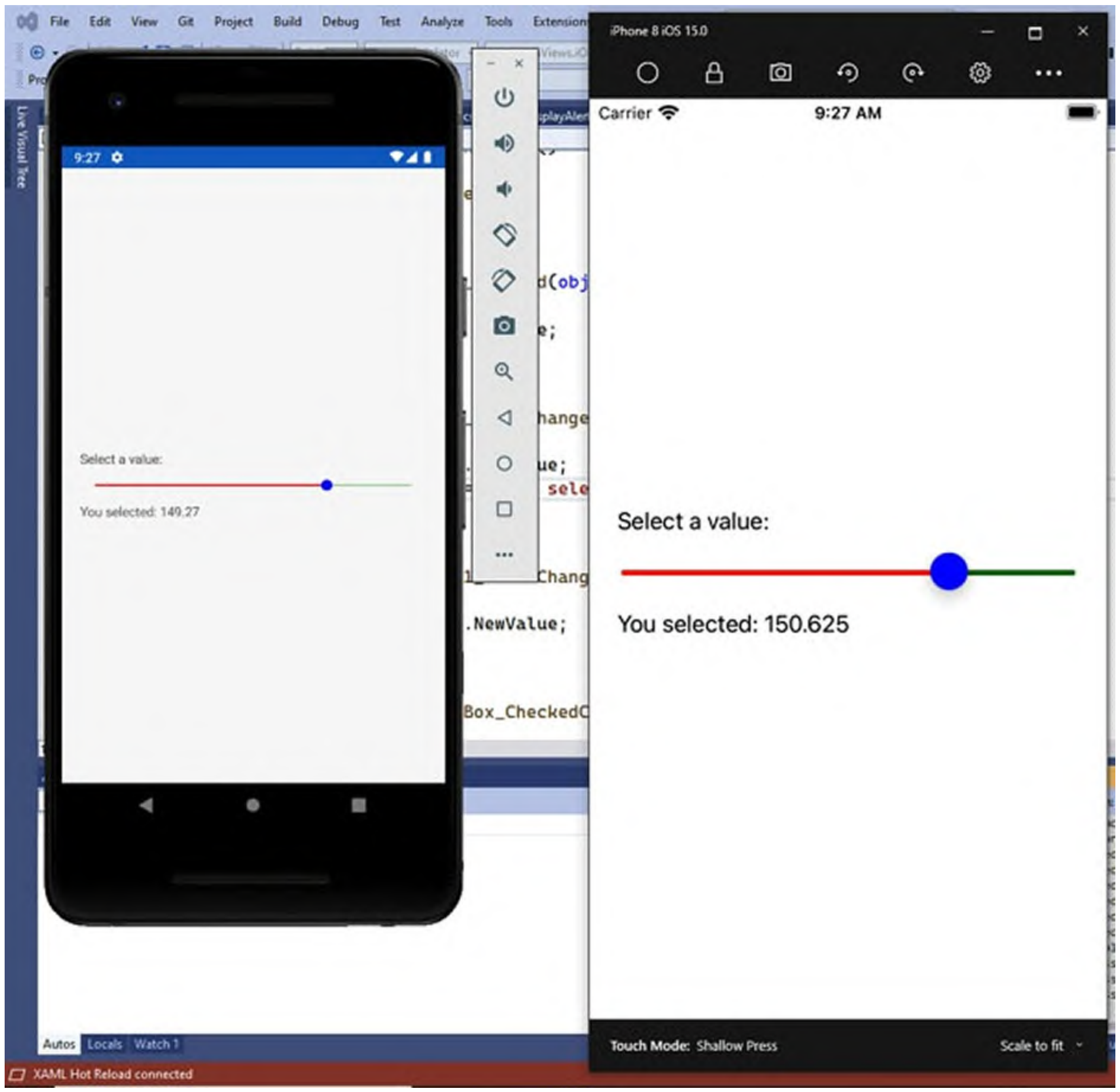


***Figure 7.10:*** *Selecting values with the Slider*

It is worth mentioning that a better way to update the **`Label`** with the value of the **`Slider`** is by binding the **`Text`** property of the **`Label`** to the **`Value`** property of the **`Slider`**; however, you will get the necessary knowledge about data binding in *Chapter 9, Resources and Data Binding*, so, for now, direct assignment is the way to do it.

## Incremental value selection with the Stepper

The **`Stepper`** is a view that allows for increasing or decreasing a value via built-in **+** and **–** buttons, by a specified increment. For better understanding, suppose you want to allow users to select their age. This can be accomplished with a **`Stepper`**, as follows:

```
<StackLayout VerticalOptions="CenterAndExpand"
    HorizontalOptions="FillAndExpand">
  <Label Text="Select your age: “/>
  <Stepper x:Name="Stepper1" Increment="1"
    Maximum="95" Minimum="13"
    Value="30" ValueChanged="Stepper1_ValueChanged"/>
  <Label Text="You selected: “ x:Name="StepperValue"/>
</StackLayout>
```

The **`Maximum`** and **`Minimum`** properties specify the numerical limits for the **`Stepper`**, whereas **`Value`** represents the currently selected value. The last **`Label`** is updated in the **`ValueChanged`** event handler, whose code is as follows:

```
private void Stepper1_ValueChanged(object sender,
ValueChangedEventArgs e)
{
  double value = e.NewValue;
  StepperValue.Text = $"You selected: {value}";
}
```

*Figure 7.11* shows the result of the preceding code:

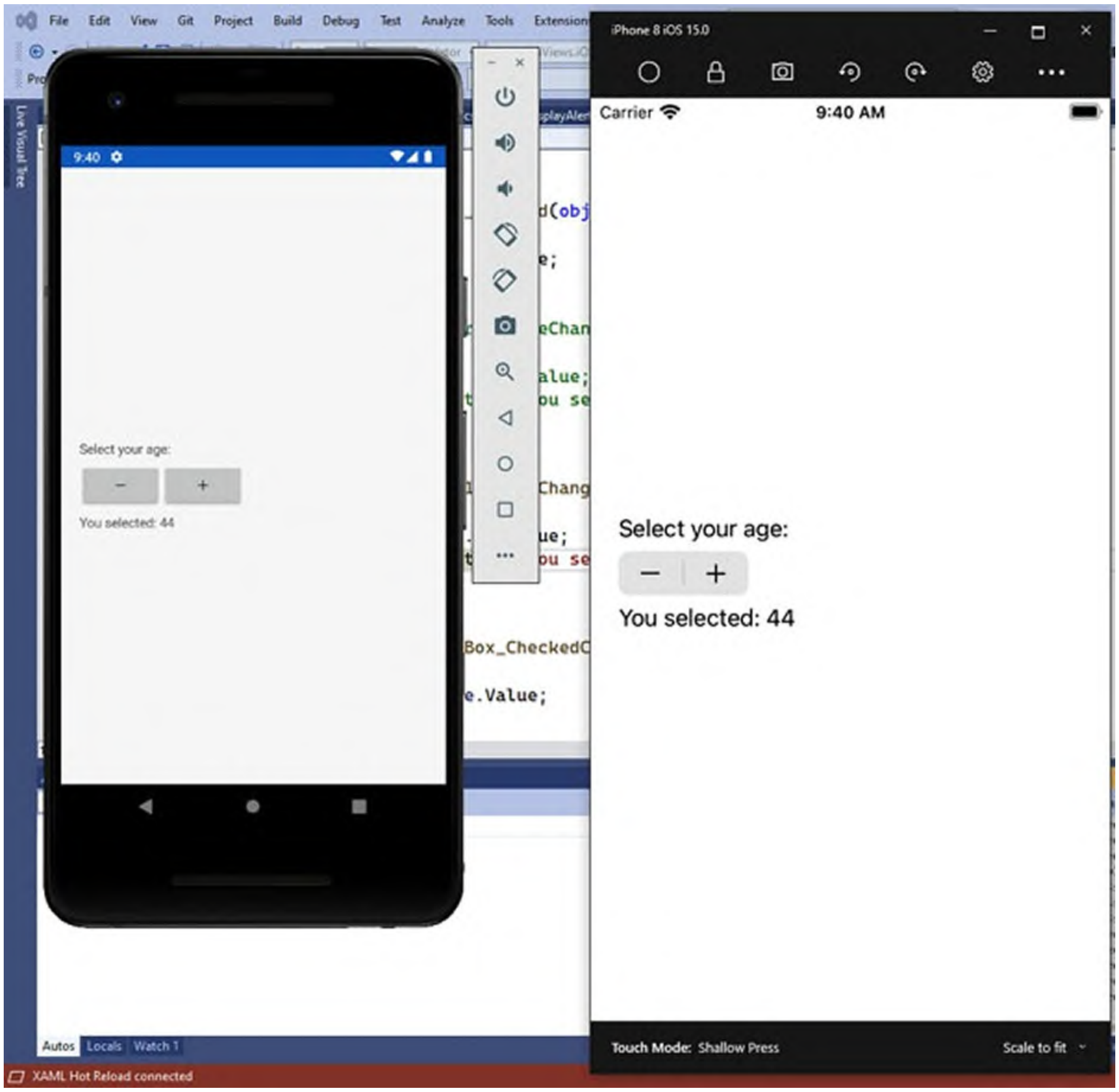


***Figure 7.11:*** *Selecting values with the Stepper*

## Implementing search functionalities

**`Xamarin.Forms`** allow you to quickly implement search functionalities in your apps with the **`SearchBar`** view. This shows the system search box, which includes the system search icon that users tap to start searching. It is worth mentioning that the **`SearchBar`** provides the user interface for searching, but the logic behind handling the search result relies completely on your work. For example, you could filter a list of elements with a LINQ query based on the entered search terms, or you could redraw the user interface to display only the visual elements that match the search criterion. You can declare a

**`SearchBar`** as follows:
```
<SearchBar x:Name="SearchBar1" Placeholder="Enter you search
key..."
    SearchButtonPressed="SearchBar1_SearchButtonPressed"
    TextChanged="SearchBar1_TextChanged" />
```
The **`SearchButtonPressed`** event is raised when the user taps the search icon, whereas **`TextChanged`** is fired at every keystroke. The logic you apply for searching should be handled when **`SearchButtonPressed`** is raised and handled by the following handler stub:
```
private void SearchBar1_SearchButtonPressed(object sender,
EventArgs e)
{
   // Implement logic based on the SearchBar’s Text property...
}
```
*Figure 7.12* shows the result of the preceding code:

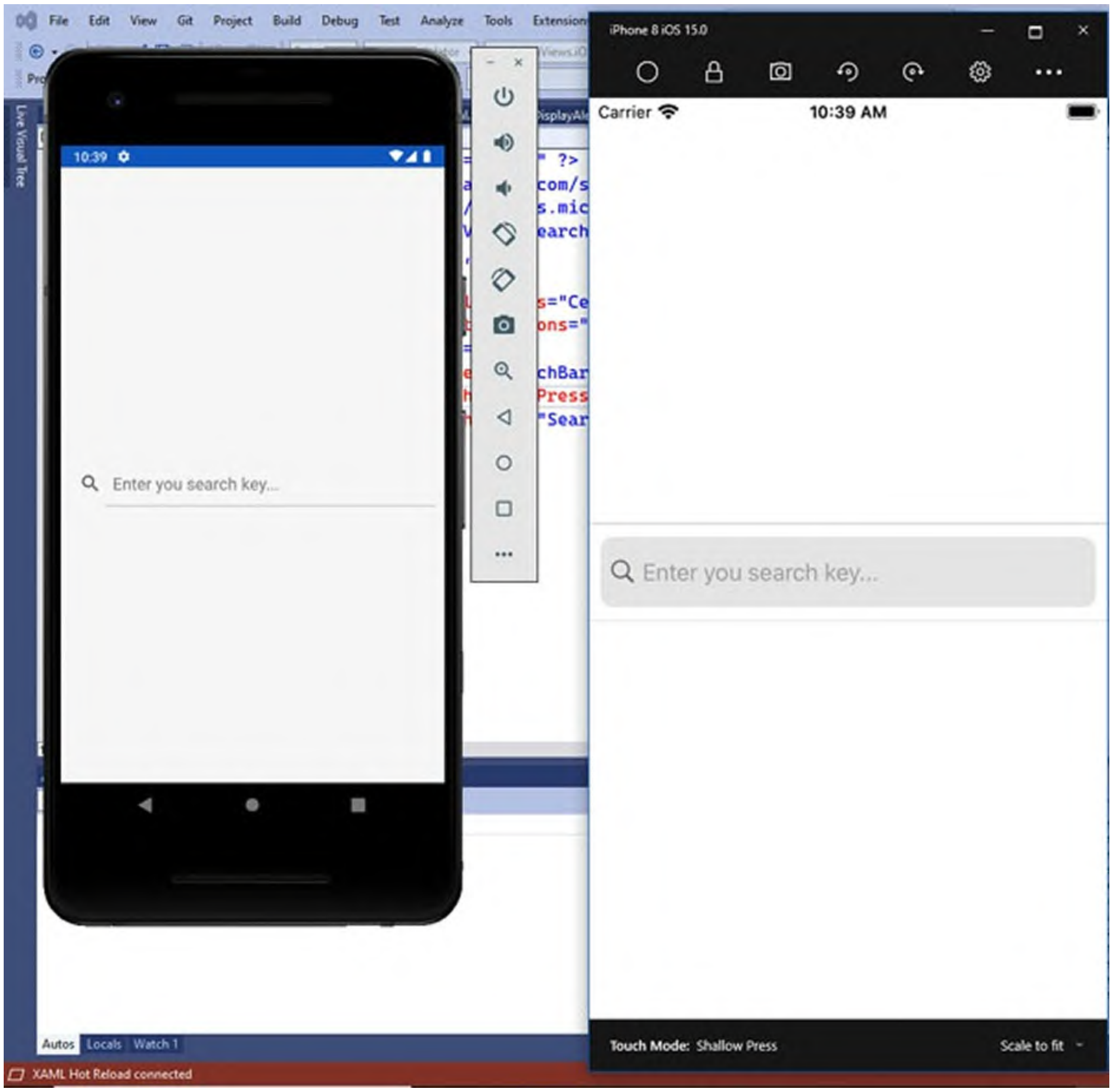


***Figure 7.12:*** *Implementing searching features*

**Tip: Behind the scenes, the `SearchBar` is an evolved `Entry`. This means you can manage all the properties related to the text in the way that was previously described for the `Entry` view, including font management and text transformations.**

In terms of customization, you can assign the **`PlaceholderColor`**, **`TextColor`**, **`BackgroundColor`**, and **`CancelButtonColor`** properties to change colors of the placeholder, search text, search bar's background, and the color of the **`Cancel`** button that is displayed inside the search bar when typing.

## Handling long-running tasks

There are operations that could last for a potentially long time, and you cannot exactly predict how long they will last. For example, downloading data from the internet, invoking Web API, complex queries over a database. In such situations, it is best practice to inform the user that a potentially long-running operation is in progress so that they are not surprised if nothing is happening in the app. For this purpose, you can leverage the **`ActivityIndicator`** view. The **`ActivityIndicator`** displays an animated spinner whose purpose is to indicate that something is in progress. Declaring and enabling an **`ActivityIndicator`** is very simple, as shown here:

```
<ActivityIndicator x:Name="ActivityIndicator1" IsVisible="true"
IsRunning="true" />
```

Actually, the spinner starts when you assign the **`IsRunning`** property with true, but you might also want to explicitly set the visibility (**`IsVisible`**) for behavior consistency across platforms. The reason why the view has a name in the previous line is that you will need to stop the indicator from running in code, for instance, with the following C# snippet:

```
private void DisableIndicator()
{
  this.ActivityIndicator1.IsRunning = false;
  this.ActivityIndicator1.IsVisible = false;
}
```

You will call this code once the long-running task has been completed. Obviously, you can also enable the **`ActivityIndicator`** programmatically whenever you need it. *Figure 7.13* shows how the progress indicator looks:

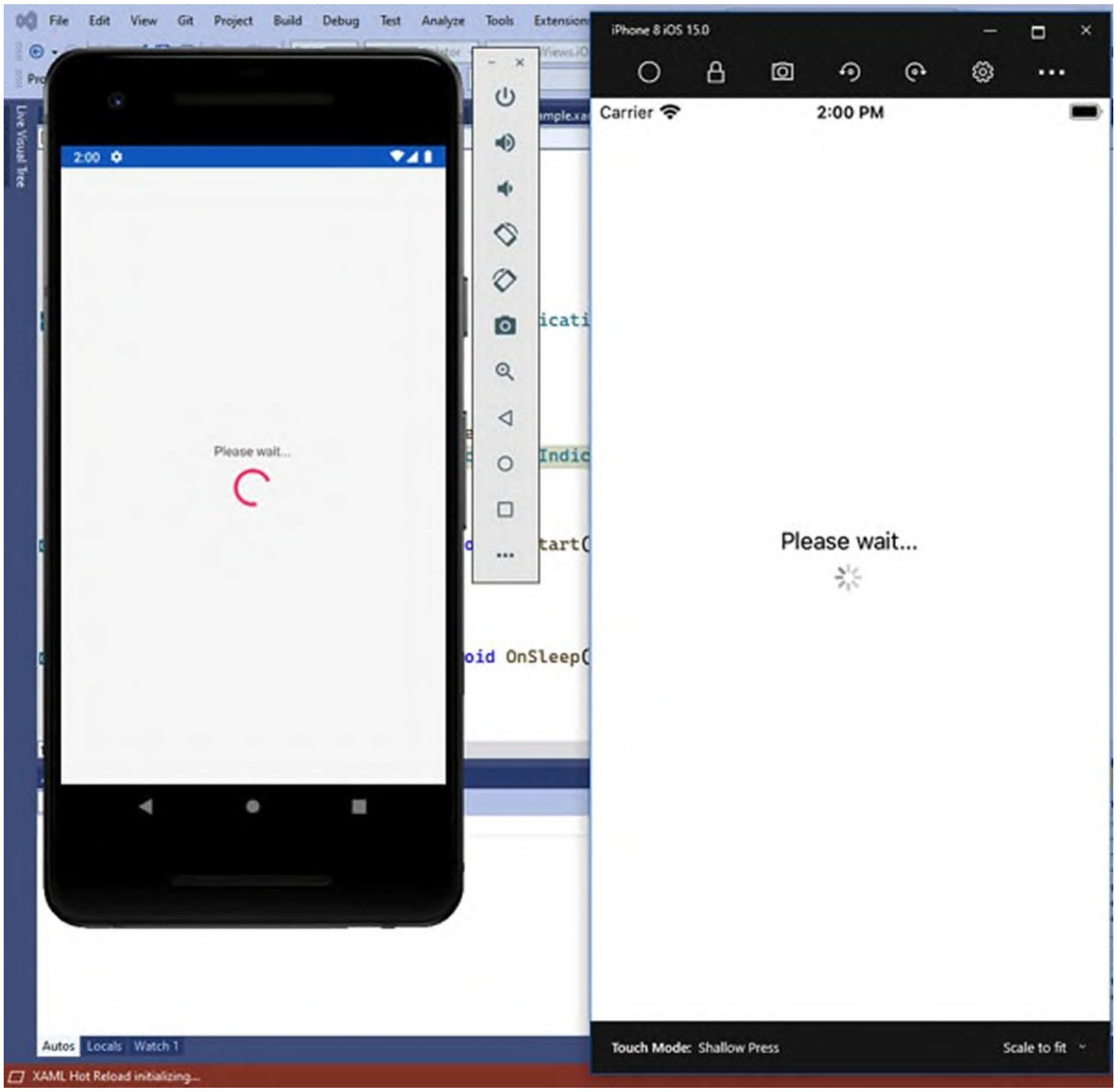


***Figure 7.13:*** *Waiting for a long-running task to complete*

The spinner look and feel is based on the system color theme, but you can assign a different color by assigning the **`Color`** property.

**Tip: Page objects, including the `ContentPage`, expose a property called `IsBusy`. When true, this displays an activity indicator on the device status bar. You might also consider this option, for example, with background tasks that do not interfere with the user interface of the app.**

## Displaying images

As a mobile app developer, one of your goals is to create beautiful and engaging apps. For this purpose, you will often include media contents, such as images, videos, and audio. Media is discussed in *Chapter 10, Brushes, Shapes, and Media*; this chapter describes how to display images. These are very common in any mobile app, and there are many reasons to use some. From a company logo to content enrichment, images are a crucial part of the user interface. Xamarin.Forms exposes a view called Image, which allows for displaying both local and remote images. Supported formats are **`.jpg`**, **`.png`**, **`.gif`** (including animated GIFs), and **`.tif`**. The path of the image you want to display is assigned to the **`Source`** property. For example, the following code displays a royalty-free picture from a remote URL (the **`Aspect`** property is described in further detail in the next section, *Handling the aspect*):

```
<Image Source="https://images.freeimages.com/images/large-
previews/72c/fox-1522156.jpg" Aspect="AspectFit"/>
```

*Figure 7.14* shows the result of this XAML. In this case, the picture is the only element of the user interface, and it is positioned at the center of the screen, but in your applications, you will certainly show images with a different arrangement:

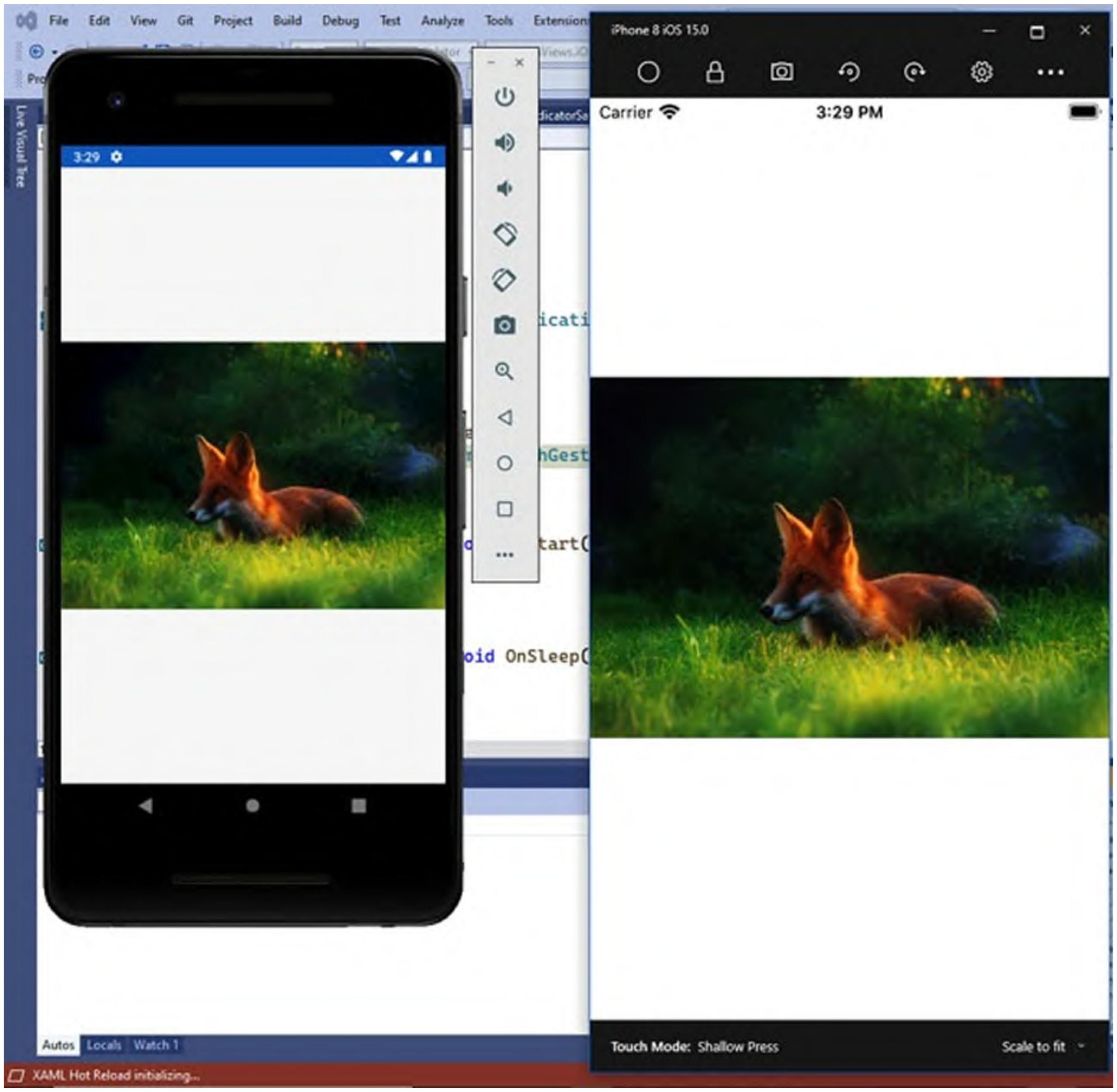


***Figure 7.14:*** *Displaying images*

The **`Source`** property is of type **`ImageSource`**. This can receive an URL, the name of a local file, or the address of an image stored inside the app's resources. Assigning a file name is very easy, like in the following example:

```
<Image Source="SampleImage.jpg" Aspect="AspectFit"/>
```

**`Xamarin.Forms`** searches for the image file inside each platform project's resources, which means the **`Resources`** folder for the Xamarin.iOS project and the **`mipmap`** subfolders of the **`Resources`** folder in the Xamarin.Android project. As you learned in the *Managing image files* section, the same image must be supplied in different resolutions to satisfy several screen factors, and the runtime will resolve the most appropriate one. It is also common to assign the **`Source`** property in C#, for example, if you need to display different images according to the context. You can do this by invoking static methods from the **`ImageSource`** class, as follows:

```
Image1.Source = ImageSource.FromUri(new
Uri(“https://mysite.com/myimage.jpg"));
Image1.Source = ImageSource.FromFile(“MyImage.jpg");
Image1.Source =
ImageSource.FromResource(“ProjectName.Folder.ImageFileName.jpg")
```

The **`FromUri`** method assigns a remote image as the source, and you just saw this in the previous example. The **`FromFile`** method assigns a local image file as the source. The **`FromResource`** method assigns an embedded resource as the view's source. In order to include an image as an embedded resource, you follow these steps:

1. Add the image file to the shared project. You can certainly copy the file into a subfolder.
2. Set its **`Build Action`** property with **`EmbeddedResource`**.
3. In the third line of previous code, replace **`ProjectName`** with the name of your project, **`Folder`** with the name of the subfolder where you copied the image file to (optional), and the file name.

**Embedding images in the app resources eliminates the need to manage image files multiple times and in multiple projects, but it will increase the app package size. This technique should be limited only to a strict number of situations.**

Assigning the **`Image.Source`** property in XAML with an embedded resource is very tricky, unless you have strong knowledge of XAML and resources in Xamarin, which you might not have at the moment. For this reason, the best and simplest approach is using the **`ImageSource.FromResource`** method in C#.

## Handling the aspect

The **`Aspect`** property specifies the size and stretching of an image within the space it occupies. You assign one of the values from the **`Aspect`** enumeration:

- **`Fill`**: The image is stretched to fill the display area exactly and completely.

- **`AspectFill`**: The image is clipped to fill the display area while keeping the original aspect.
- **`AspectFit`**: Makes the entire image fit into the display area. If the image is not wide or tall enough, some blank space is also added to the sides.

It is also possible to adjust an image width and height via the **`WidthRequest`** and **`HeightRequest`** properties.

## Managing image files

Adding images as local files is platform-specific, so the behavior is different for iOS and Android. On iOS, the same image must be supplied in three different resolutions. Supposing you have an image file called **`MyImage.png`**, you will need the following files: **`MyImage.png`**, **`MyImage@2x.png`**, and **`MyImage@3x.png`**. In the real world, it is the responsibility of a professional designer to export an image into the required file formats (and discussing design tools that can do this is out of the scope of this book), so you do not need to worry about generating them on your own. Examples of tools that are capable of exporting images into iOS and Android assets are Adobe Xd and Figma (**https://www.figma.com**). Files must be copied into the **`Resources`** folder of the Xamarin.iOS. Visual Studio should automatically set the **`Build Action`** property of each file with **`BundleResource`**, but make sure you double-check this. For Android, you will need six different resolutions. If you expand the **`Resources`** folder of the Xamarin.Android project in Visual Studio, you will see the following subfolders:

- mipmap-anydpi-v26
- mipmap-hdpi
- mipmap-mdpi
- mipmap-xhdpi
- mipmap-xxhdpi
- mipmap-xxxhdpi

Your designer will supply files in the appropriate resolution, which you will copy, from the lowest to the highest, into the aforementioned subfolders in the order that they have been mentioned. By the way, the Microsoft documentation has a page about managing images in Xamarin.Forms solutions, which you are encouraged to read: **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/images**.

## Adding interactivity to Views

Not all the views in **`Xamarin.Forms`** natively support user interaction. For example, you cannot tap a **`Label`**, but sometimes you might want to provide users the option to tap a view that is not a button, or you might want to ask for the user confirmation before a task starts. This section describes how to add interactivity to views that do not support this option directly.

## Implementing GestureRecognizers

Views that do not support user interaction directly, such as all the layouts or the **`Label`** and the **`Image`**, expose the **`GestureRecognizers`** property, of type **`IList<GestureRecognizer>`**. The **`GestureRecognizer`** object is the base class for the following gesture recognizers:

- **`TapGestureRecognizer`**: This adds tap capabilities to a view.
- **`PinchGestureRecognizers`**: This adds support for detecting the movement of fingers on screen.
- **`PanGestureRecognizers`**: This adds support for dragging objects.
- **`SwipeGestureRecognizer`**: This adds tap capabilities to a view.
- **`DragGestureRecognizers`**: This adds support for the pinch-to-zoom gesture.

As a Xamarin jobseeker focusing on general purpose code, the one you really need to know is the **`TapGestureRecognizer`**, which is certainly the most used gesture inside real-world apps. Though useful, the other gestures are less used, and in most cases, there are more recently added views that implement the same gestures (such as the **`SwipeView`**). The following code shows how to add tap support to a **`Label`** view using a **`TapGestureRecognizer`**:

```
<Label Text="Tap Here!" HorizontalOptions="CenterAndExpand"
    VerticalOptions="CenterAndExpand">
  <Label.GestureRecognizers>
    <TapGestureRecognizer x:Name="LabelTap"
    NumberOfTapsRequired="1" Tapped="LabelTap_Tapped"/>
  </Label.GestureRecognizers>
</Image>
```

The **NumberOfTapsRequired** property allows you to specify how many taps are required to enable interaction, whereas the **Tapped** event is raised when the user taps the view. The event handler looks like the following:

```
private void LabelTap_Tapped(object sender, EventArgs e)
{
  // The Label has been tapped
}
```

If you wish to read more about the other gesture recognizer, the documentation has a dedicated page available at **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/gestures**. In this book, you will find details about specific views that you can use to accomplish the same kinds of gestures in a more structured way.

## Displaying and handling alerts

Alerts are small system dialogs that are used to send messages to the user and ask for user confirmation about an operation that requires attention before it is executed. In **Xamarin.Forms**, every **Page** object exposes an asynchronous method called **DisplayAlert** that you can invoke for this purpose. The following code shows how to display a one-way alert when a button is tapped:

```
private async void Button1_Clicked(object sender, EventArgs e)
{
  await DisplayAlert(“Warning", “The battery level is low",
  “OK");
}
```

The first parameter of the method is the title for the dialog, the second parameter is the content, and the third parameter is the text for the only button that appears. In addition, an overload of **DisplayAlert** can be used to accept the user input. The following code demonstrates this:

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

The third and fourth parameters of the method represent confirmation and cancellation options, and you can add any text you like for these. If the user selects the confirmation option, **DisplayAlert** returns true; otherwise, it returns false. *Figure 7.15* shows the result of the code:

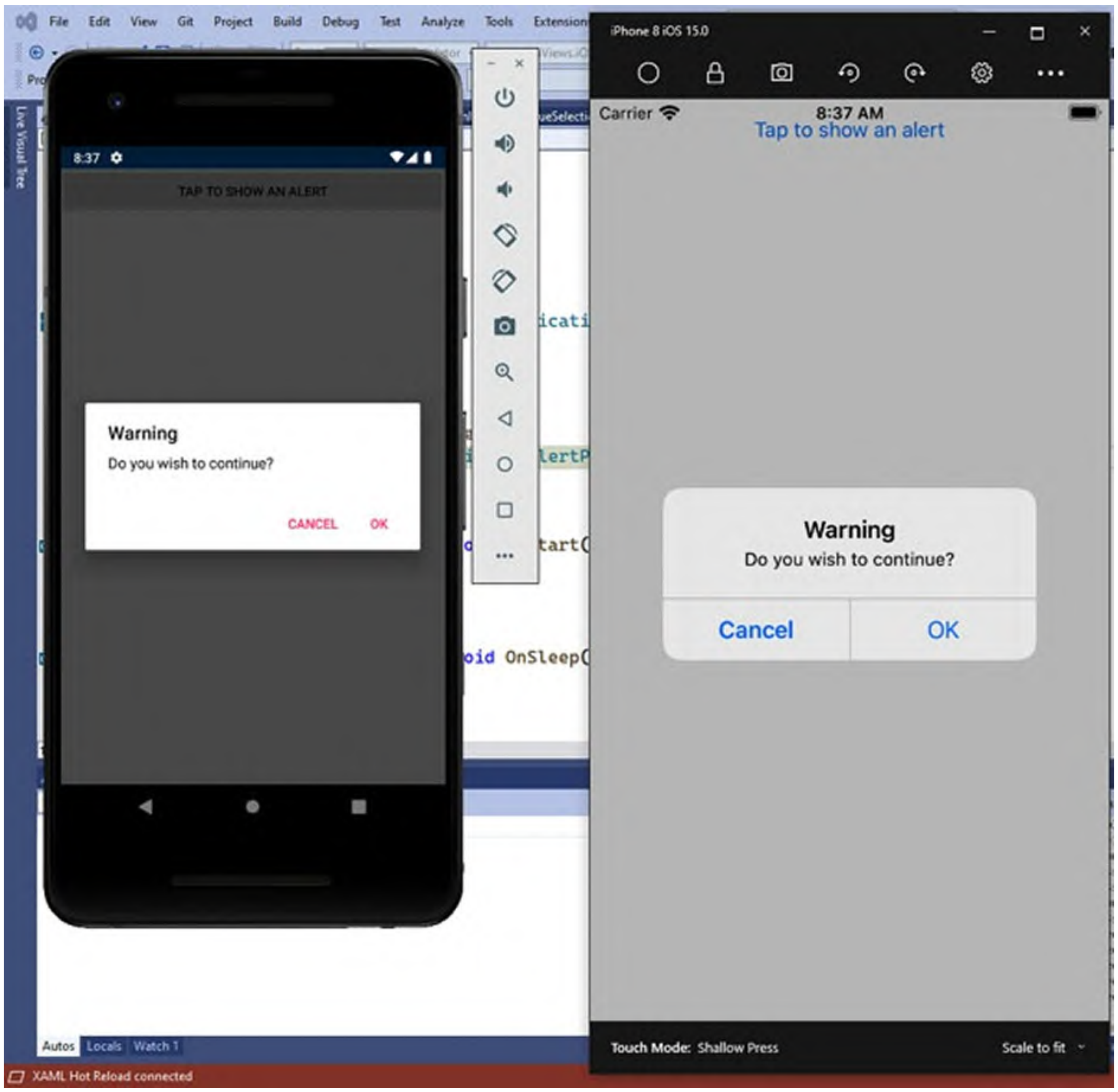


***Figure 7.15:*** *Displaying alerts*

## Understanding visual states

Views have states that describe their current condition. For example, a view can have a **`Normal`** state, **`Disabled`** state, or **`Focused`** state. You can change the appearance of a view depending on its state in C# code by implementing your logic, or you can take advantage of a feature called Visual State Manager. This feature allows you to change the appearance of a view, based on its state, completely in XAML. For instance, you can change some of a view's colors depending on the state. The following XAML shows how to change the background color of an **`Entry`**, according to the state:

```
<Entry VerticalOptions="Center"
HorizontalOptions="FillAndExpand"
   Placeholder="Enter some text here...">
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="CommonStates">
      <VisualState x:Name="Normal">
        <VisualState.Setters>
          <Setter Property="BackgroundColor" Value="White" />
        </VisualState.Setters>
      </VisualState>
      <VisualState x:Name="Focused">
        <VisualState.Setters>
          <Setter Property="BackgroundColor"
            Value="LightBlue" />
        </VisualState.Setters>
      </VisualState>
      <VisualState x:Name="Disabled">
        <VisualState.Setters>
          <Setter Property="BackgroundColor" Value="Gray" />
        </VisualState.Setters>
      </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
</Entry>
```

The **`VisualStateManager`** object exposes a **`VisualStateGroups`** collection, where you place a root object that contains as many **`VisualState`** objects as the number of states you want to represent. For each **`VisualState`**, you specify a **`VisualState.Setters`** group, where you change the value of the properties of your interest, on a given state. In this example, the code is only setting the **`BackgroundColor`** property, and this is done via an object called **`Setter`** that receives the property name and its new value. *Figure 7.16* shows, from left to right, how the **`Entry`** appears based on its state. For the **`Disabled`** state, you usually enable or disable views depending on your own logic, but for the current example, you can explicitly assign the **`IsEnabled`** property with false:

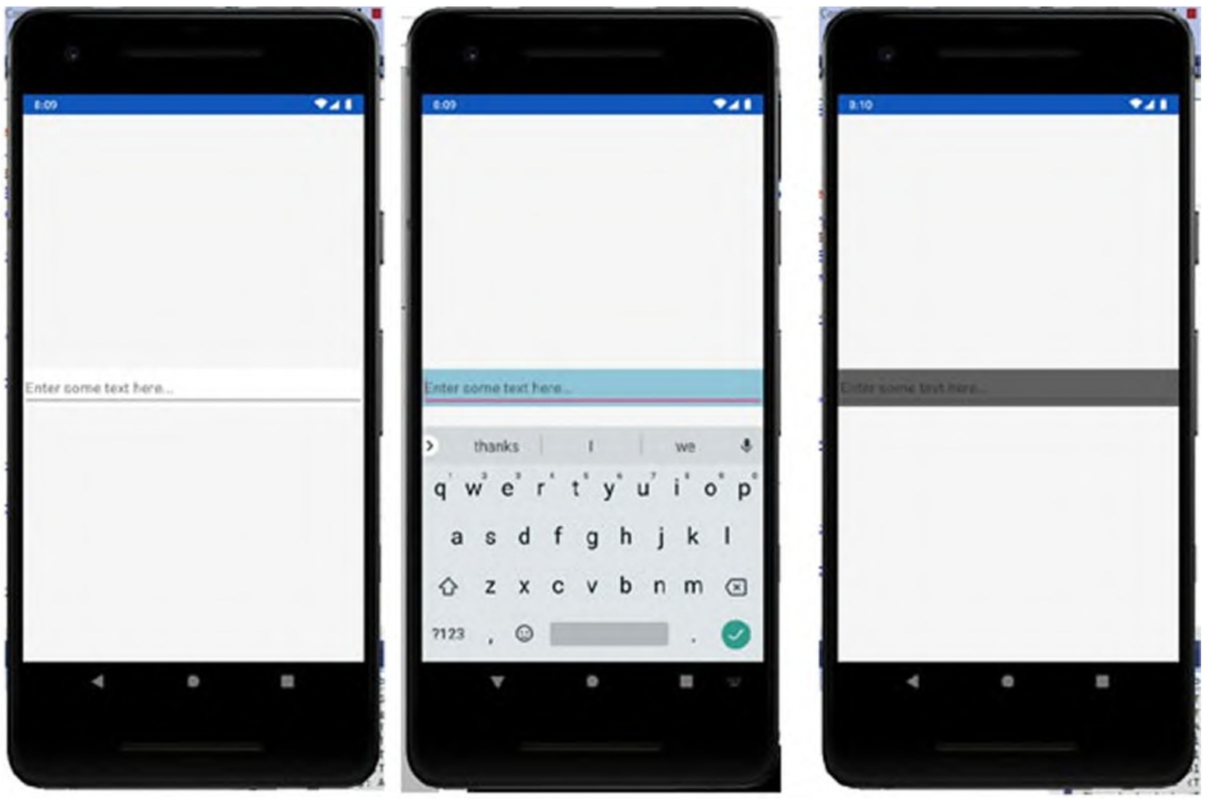


***Figure 7.16:*** *Managing visual states: Normal, Focused, Disabled*

The **`Normal`**, **`Disabled`**, **`Focused`**, and **`Selected`** states are common to every view in **`Xamarin.Forms`**. Some additional, specific states are available to the views listed in *Table 7.5*:

| View | Specific visual states |
|---|---|
| **`Button`** | **`Pressed`** |
| **`CheckBox`** | **`IsChecked`** |
| **`CarouselView`** | `DefaultItem`, `NextItem`, `PreviousItem`, `CurrentItem`. (The `CarouselView` is discussed in *Chapter 9, Resources and Data Binding*). |
| **`ImageButton`** | **`Pressed`** |
| **`RadioButton`** | `Checked` and `Unchecked` |
| **`Switch`** | `On` and `Off` |

***Table 7.5:*** *Specific visual states*

The Visual State Manager is certainly useful to developers, but its real power is in allowing professional designers to work on visual states all in declarative code without the need for implementing C# logic.

## Conclusion

Views are the building blocks of the user interface, which you implement to add functionalities to your applications. This chapter described the so-called common views and primitive controls that allow for the most common user interaction and input, such as displaying and entering text, working with dates and time, selecting values, and enriching the UI with images. You saw views working inside an individual page, but in the real world, this is not certainly how mobile apps work. The next chapter will describe another important part of the user interface implementation, which is navigating between pages.

## Key terms

- **View**: An individual primitive control that implements a functionality in your app.
- **Typeface**: The synonym of font, it represents the size, weight, and style of text.
- **Content Page**: An individual page in a `Xamarin.Forms` shared project.
- **Gesture recognizer**: An object that identifies finger gestures on screen.
- **Visual state**: The condition of a view in a specific moment of its lifecycle.

## Suggested readings

Microsoft official Xamarin.Forms reference for the user interface (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface**).

# CHAPTER 8
# Pages and Navigation

## Introduction

So far, you have seen how to implement layouts and views inside a single, blank page, represented by a **`ContentPage`** object. However, mobile applications rely on different kinds of pages, which allow for different user experiences on multiple pages. This chapter explains the available page types in **`Xamarin.Forms`** and how they work cross-platform, and it also explains how to implement navigation between pages, including a common app infrastructure represented by the Shell. The approach will be based on the point of view of the Xamarin jobseeker, which means focusing on new projects while keeping an eye on what you might encounter with existing applications.

## Structure

In this chapter, we will cover the following topics:

- Introducing available pages
- Navigating between pages
- Common app features: The Shell

## Objectives

After completing this chapter, you will have full knowledge of the building blocks of the user interface in **`Xamarin.Forms`**, and you will be able to implement the appropriate pages and navigation structure according to the business requirements. This will also provide you with the foundation to understand the concepts described in the upcoming chapters.

## Introducing available pages

Mobile applications can have side menus and navigation bars, and contents might be organized within tabs. For the best user experience possible, **`Xamarin.Forms`** provide many pages that represent, in a cross-platform approach, how a native app displays contents. In terms of code, all the available pages inherit from the **`Page`** abstract class, and each page is a root element in the visual tree. The **`ContentPage`** is the one you have used so far to represent an individual, blank page, but there are five pages available in **`Xamarin.Forms`**, as described in *Table 8.1*.

| Page type | Description |
|---|---|
| **`ContentPage`** | Represents an individual page. |
| **`FlyoutPage`** | Represents a page with master-details view. |
| **`TabbedPage`** | Groups multiple `ContentPage` pages by tabs. |
| **`CarouselPage`** | Allows for swiping between `ContentPage` objects. |
| **`NavigationPage`** | Implements the infrastructure for navigating between pages. |

***Table 8.1:*** *Page types in Xamarin.Forms*

As a general rule, every page can contain only one root visual element, typically, a layout that is then articulated with a more sophisticated implementation of the user interface based on child layouts. This section describes the available page types, explaining when it is best to use them.

## Individual pages: The ContentPage

You already used the **`ContentPage`** many times in the previous chapter. This page not only works as a standalone page but can also be the child element for other pages, as you see in this chapter. For this reason, it is important to underline its **`Title`** property, which identifies the currently active page within parent pages, like the **`TabbedPage`** or the **`CarouselPage`**. In addition, it is useful to know that the **`Content`** tag can be omitted. Based on this, the following code is completely legal (also note the **`Title`** property declaration):

```
<?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:IntroducingPages"
    Title="Main page"
    x:Class=" IntroducingPages.MainPage">
  <Label Text="A content page with title"/>
```

```
</ContentPage>
```

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

## Master-details views: The FlyoutPage

You have certainly used mobile apps that implement a side-menu, typically on the left of the screen, that you can open or close with a swipe gesture and that allows for selecting one item from multiple options, and when you select an item, specific content is shown on the right side of the screen. This kind of user interface is also known as **master-details**. In the previous versions of **Xamarin.Forms**, a page called **MasterDetailPage** was available, but then it was made obsolete in favor of a new page called **FlyoutPage**. This allows for implementing a master-details view in a more modern way, starting from the terminology. In fact, the flyout is the side menu that can be shown or hidden by the user. For better understanding, consider *Figure 8.1*, where you can see the flyout:

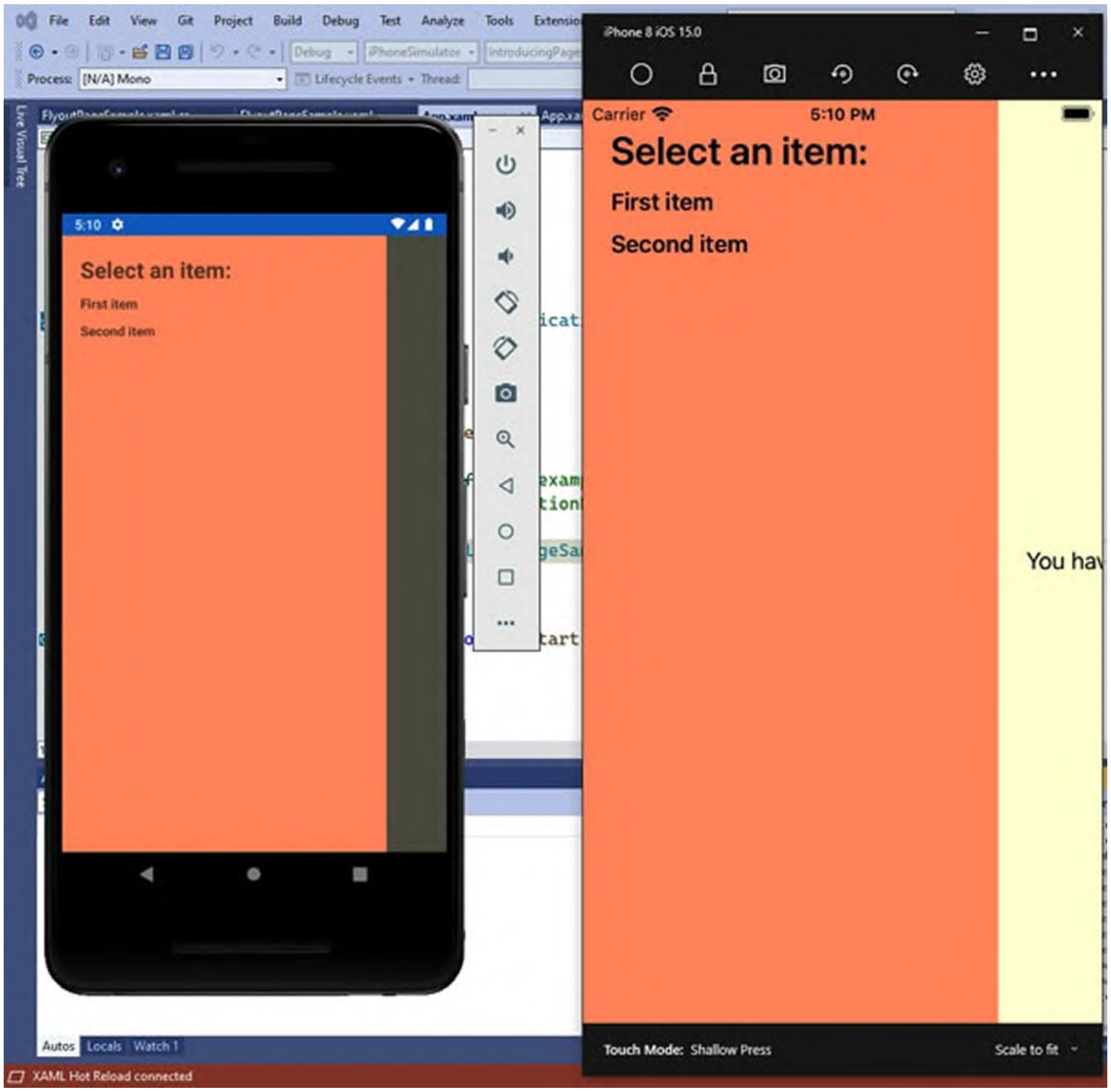


***Figure 8.1:*** *The flyout in a FlyoutPage*

*Figure 8.2* shows you its details:

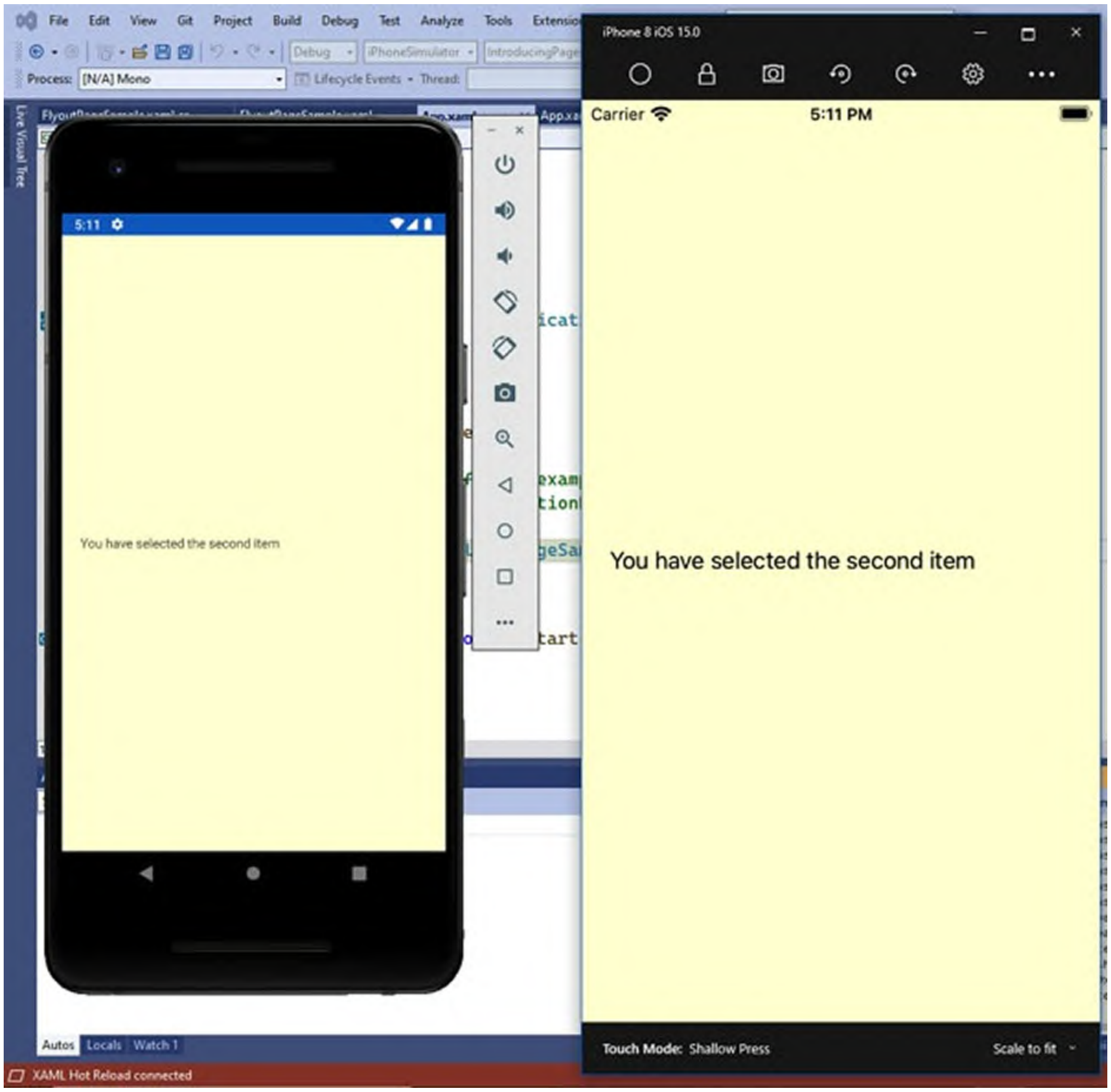

You have selected the second item
Carrier
5:11 PM
iPhone 8 iOS 15.0
Touch Mode: Shallow Press
Scale to fit
XAML Hot Reload connected

***Figure 8.2:*** *The details in a FlyoutPage*

In both figures, you can see an example based on a list of items in the flyout and the details content changes depending on the selection. The following code demonstrates how to replicate the result you see in the figures, and it also represents the idea behind the flyout:

```
<?xml version="1.0" encoding="utf-8" ?>
<FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="IntroducingPages.FlyoutPageSample">
  <FlyoutPage.Flyout>
   <ContentPage Title="Main page" BackgroundColor="LightSalmon">
    <StackLayout Spacing="10" Margin="0,20,0,0">
```

```
      <Label Text="Select an item:" FontSize="Title"
        Margin="20,0,0,0" FontAttributes="Bold"
        HorizontalOptions="Start"/>
      <Label Text="First item" x:Name="Item1" Margin="20,0,0,0"
          FontAttributes="Bold">
        <Label.GestureRecognizers>
          <TapGestureRecognizer x:Name="Item1Tapped"
              Tapped="Item1Tapped_Tapped"/>
        </Label.GestureRecognizers>
      </Label>
      <Label Text="Second item" x:Name="Item2" Margin="20,0,0,0"
          FontAttributes="Bold">
        <Label.GestureRecognizers>
          <TapGestureRecognizer x:Name="Item2Tapped"
              Tapped="Item2Tapped_Tapped"/>
        </Label.GestureRecognizers>
      </Label>
    </StackLayout>
  </ContentPage>
 </FlyoutPage.Flyout>
 <FlyoutPage.Detail>
  <ContentPage BackgroundColor="LightYellow">
    <StackLayout VerticalOptions="Center"
    HorizontalOptions="Start">
      <Label Margin="20,0,0,0"
          Text="You have selected the first item"
          x:Name="Item1Details"/>
      <Label Margin="20,0,0,0"
          Text="You have selected the second item"
          x:Name="Item2Details"
          IsVisible="False"/>
    </StackLayout>
  </ContentPage>
 </FlyoutPage.Detail>
</FlyoutPage>
```

Here's a list of relevant points about the **`FlyoutPage`** and the preceding code:

- The **`FlyoutPage.Flyout`** object allows you to implement the flyout.
- The content of the **`Flyout`** is typically a **`ContentPage`** but this is not a rule; it can be any object deriving from **`Page`**.
- When the user opens the flyout, the **`IsPresented`** property from the **`FlyoutPage`** is set to true. When it is hidden, **`IsPresented`** is set with false. You can programmatically control the flyout by assigning **`IsPresented`** as per your convenience.

- Following from the previous point, when the user changes the visibility of the flyout, an event called **`IsPresentedChanged`** is also fired.
- In the example, a list of items is supplied with a **`StackLayout`** that contains a few **`Label`** views. Interaction is enabled via the **`TapGestureRecognizer`** object.
- The **`Flyout.Detail`** object implements the details part of the **`FlyoutPage`**. The visual tree of the details is made of a **`StackLayout`** with two **`Label`** views, whose visibility changes depending on the user selection in the flyout.

The interaction with labels in the flyout and the visibility change for labels in the details is handled by the following C# code:

```
private void Item1Tapped_Tapped(object sender, EventArgs e)
{
  Item1Details.IsVisible = true;
  Item2Details.IsVisible = false;
}
private void Item2Tapped_Tapped(object sender, EventArgs e)
{
  Item1Details.IsVisible = false;
  Item2Details.IsVisible = true;
}
```

For the sake of clarity, this example is working with a **`ContentPage`** in the details part, and with only one **`ContentPage`**. However, the details could be any object deriving from **`Page`**. In addition, instead of handling the visibility of individual views, you could assign a different **`Page`** object to the **`Flyout`** property. For instance, suppose you had a **`ContentPage`** called **`CustomerListPage`**; this could be assigned to the detail as follows:

```
Detail = new CustomerListPage();
```

Obviously, you are not limited to implement item selection in the flyout because this contains a page where you can implement your own user experience. Later in the chapter, when talking about the Shell, you will see a different approach to build a flyout.

**Tip: Specifying and assigning the `Title` property on the page assigned to the `Flyout` is mandatory; otherwise, an exception is thrown.**

The behavior of the flyout can also be controlled via a property called **`FlyoutLayoutBehavior`** property. *Table 8.2* describes the supported values.

| Value | Description |
|---|---|
| **`Default`** | Both the flyout and detail parts are rendered based on each platform's default layout. |
| **`Popover`** | Forces the flyout to cover the detail. |
| **`Split`** | Both the flyout and detail have equal size. |
| **`SplitOnLandScape`** | Similar to `Split`, but it is applied only when the device is in landscape orientation. |
| **`SplitOnPortrait`** | Similar to `Split`, but it is applied only when the device is in portrait orientation. |

***Table 8.2:*** *Changing the Flyout behavior*

## Organizing contents within tabs: The TabbedPage

**`Xamarin.Forms`** provides an easy way to organize contents within tabs, represented by the **`TabbedPage`** object. The **`TabbedPage`** allows you to implement several tabs, where each tab contains a **`ContentPage`** for simpler navigation. *Figure 8.3* shows an example of **`TabbedPage`**:

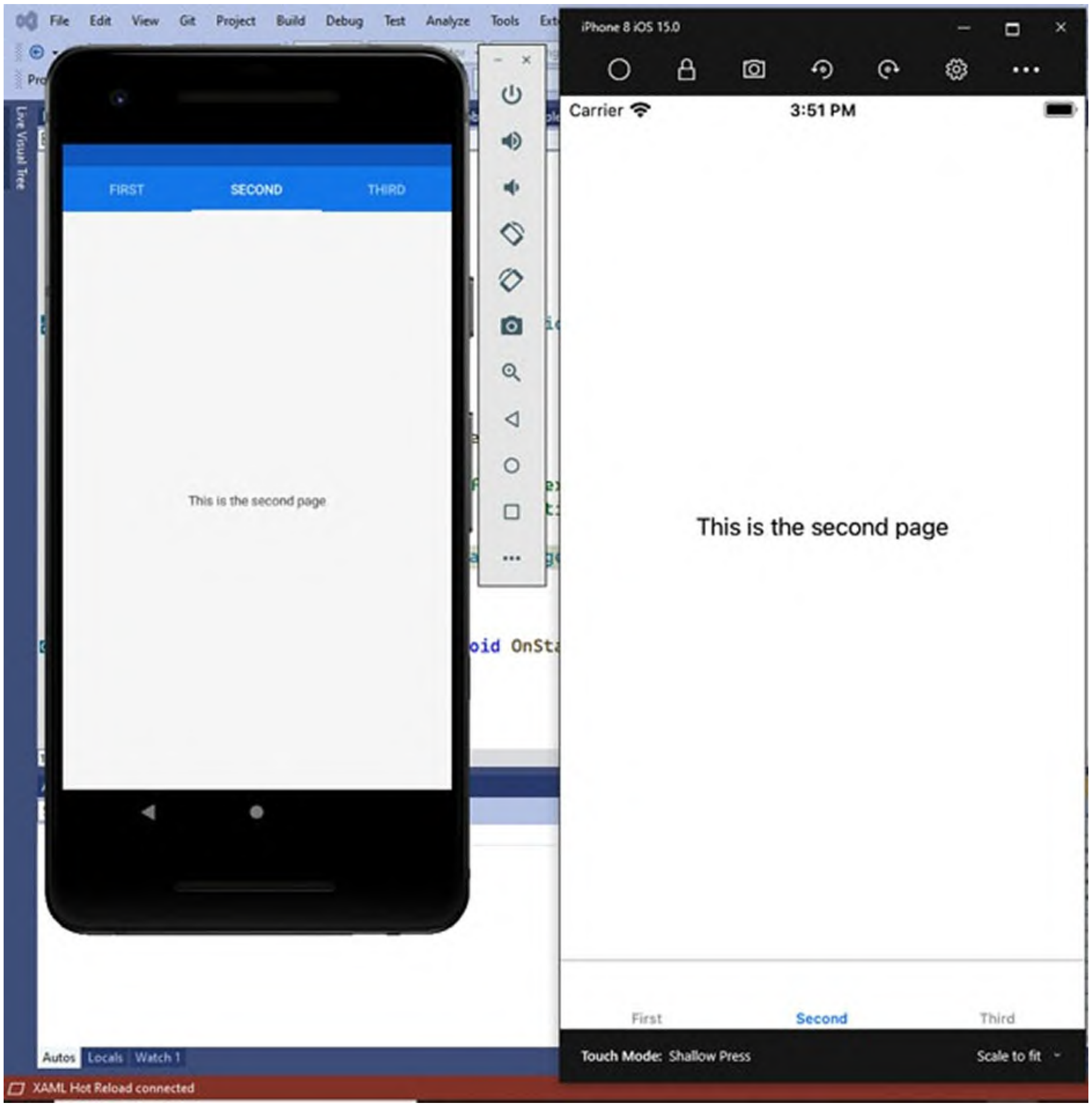


***Figure 8.3:*** *Organizing contents within a TabbedPage*

The **`TabbedPage`** can be considered as a container of **`Page`** objects, where each one can be accessed by selecting a tab. If you look at *Figure 8.3*, you can see how tabs are rendered according to the target system's design guidelines, which means at the top on Android devices and at the bottom on iOS devices. The good news is that you do not need to handle navigation between tabs because it is a built-in feature. You just need to supply the content pages. The following code demonstrates how to build the user interface you see in *Figure 8.3*:

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="IntroducingPages.TabbedPageSample">
  <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>
```

There are two key points to underline:

- The **`TabbedPage.Children`** collection must be populated with **`Page`** objects, in this case, **`ContentPage`** objects.
- Each **`ContentPage`** must specify the **`Title`** property, which is also the text displayed on the corresponding tab.

There is no limit to the number of tabs you can add, but the recommendation is no more than three or four, especially if your app will work on smaller screens (like phones).

## Scrolling pages: The CarouselPage

One of the most common gestures in the user experience of mobile apps is scrolling content with an action called **swiping**. An example could be swiping through a gallery of pictures. In **`Xamarin.Forms`**, there are views that allow for swiping their own child contents, but you can also do so with pages using the **`CarouselPage`** object. For better understanding, consider the example shown in *Figure 8.4*:

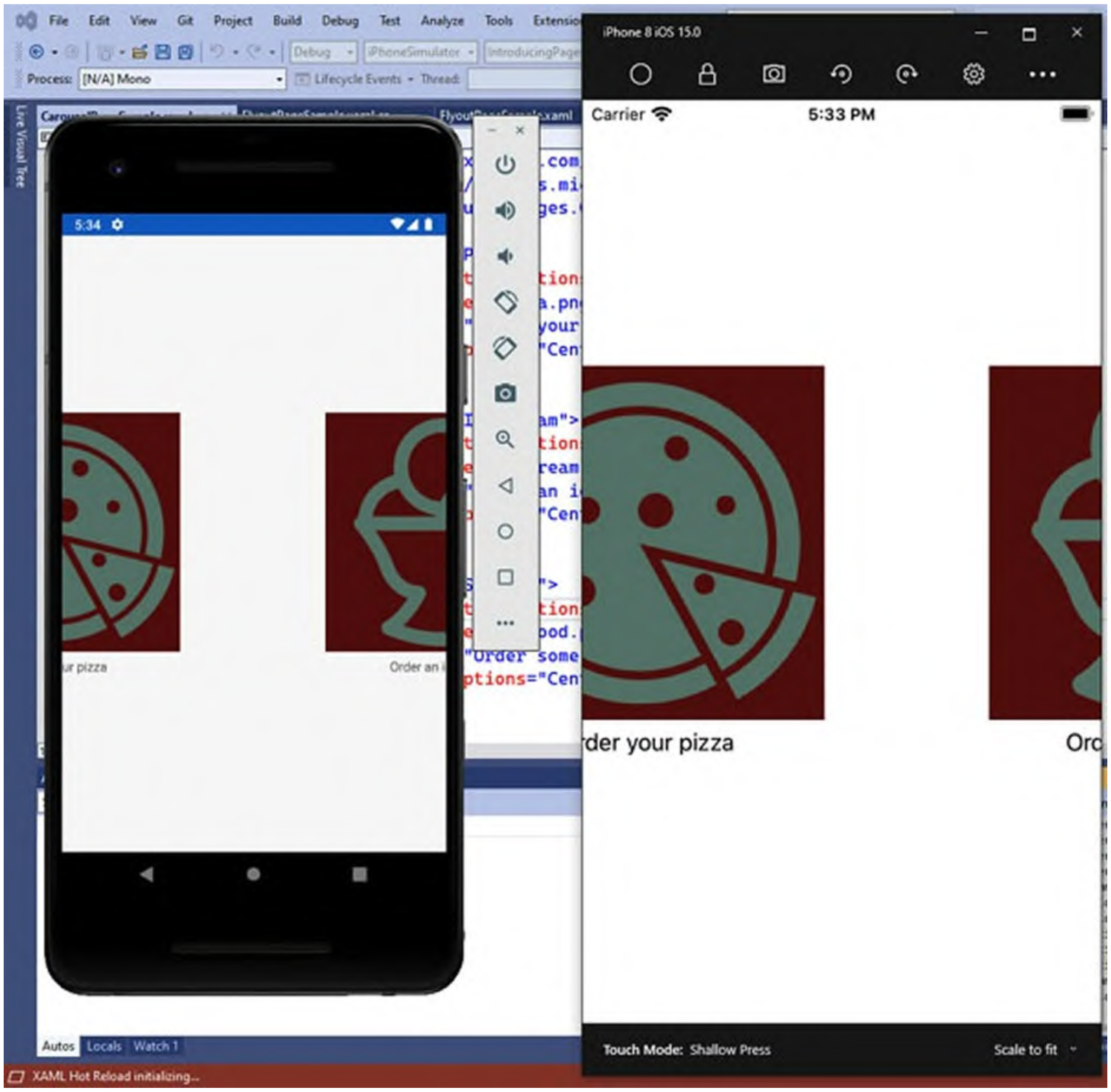


***Figure 8.4:*** *Swiping contents with the CarouselPage*

The example in [*Figure 8.4*]() shows a **`CarouselPage`** while swiping is happening. When the swipe gesture is completed, the selected **`ContentPage`** gets the focus. You can swipe both left and right. The way you declare a **`CarouselPage`** is just like you saw with the **`TabbedPage`**. In fact, you still need to populate the **`Children`** collection with **`ContentPage`** objects. The following code demonstrates how to produce the preceding example:

```
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="IntroducingPages.CarouselPageSample">
  <CarouselPage.Children>
   <ContentPage Title="Pizza">
```

```
      <StackLayout VerticalOptions="CenterAndExpand">
        <Image Source="pizza.png"/>
        <Label Text="Order your pizza" HorizontalOptions="Center"
          VerticalOptions="Center"/>
      </StackLayout>
    </ContentPage>
    <ContentPage Title="Ice cream">
      <StackLayout VerticalOptions="CenterAndExpand">
        <Image Source="icecream.png" />
        <Label Text="Order an ice cream" HorizontalOptions="Center"
          VerticalOptions="Center"/>
      </StackLayout>
    </ContentPage>
    <ContentPage Title="Seafood">
      <StackLayout VerticalOptions="CenterAndExpand">
        <Image Source="seafood.png" />
        <Label Text="Order some seafood" HorizontalOptions="Center"
          VerticalOptions="Center"/>
      </StackLayout>
    </ContentPage>
  </CarouselPage.Children>
```

**Tip: The images used in the sample code can be found in the companion solution, under the `Resources` folder of the native projects.**

You can then add actions and handle events in C# according to the structure of your user interface; for example, adding buttons or gesture recognizers.

## Navigating between pages

More often than not, mobile applications are based on multiple pages, and each page offers specific contents. Generally speaking, this is achieved by implementing page navigation. **`Xamarin.Forms`** makes it easy to implement page navigation via a built-in navigation framework exposed, at the highest level, by a special page called **`NavigationPage`**. **`Xamarin.Forms`**’ navigation framework contains a stack of pages; when a new page is opened, it is added to the stack, and when it is closed, it is removed from the stack, in a **Last-In, First-Out** (**LIFO**) approach. Practically speaking, you typically wrap the root page of your use interface inside an instance of the **`NavigationPage`** object, in the **`App.xaml.cs`** file, as follows:

```
public App()
{
  InitializeComponent();
  // NavigationSample is a ContentPage object
  MainPage = new NavigationPage(new NavigationSample());
}
```

This could also be done in XAML, but the **`NavigationPage`** is the only page that is intended to be normally used in C#. The **`NavigationPage`** enables the following:

- The navigation stack in the built-in navigation framework.
- The soft navigation bar on all the supported operating systems (with the exception of modal pages described shortly).
- The physical back button on Android. While this is always enabled, instead of closing an app, it will make the app go back to the previous page with navigation.

From the root page in the navigation stack, you can navigate to a second page by invoking the **`PushAsync`** method:

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

The **`Push`** prefix stands for pushing to the stack. From the secondary page, you can go back to the previous page by invoking the **`PopAsync`** method, as follows:

```
// SecondaryPage is popped from the stack and the app goes back
to the previous page
await Navigation.PopAsync();
```

The **`Pop`** prefix stands for popping from the stack. As an alternative, you could use the **`PushModalAsync`** and **`PopModalAsync`** methods to implement modal pages:

```
await Navigation.PushModalAsync(new SecondaryPage());
await Navigation.PopModalAsync();
```

**Tip: In short, modal pages take the full screen and do not enable the navigation bar. Use them when you want to have the highest level of control over user actions against page navigation.**

The aforementioned methods are invoked over a property called **Navigation**, which is exposed by every **`Page`** object and is the .NET representation of the navigation stack. *Figure 8.5* shows the result of the code in the companion solution. Note how the navigation bar is also enabled on Android and iOS:

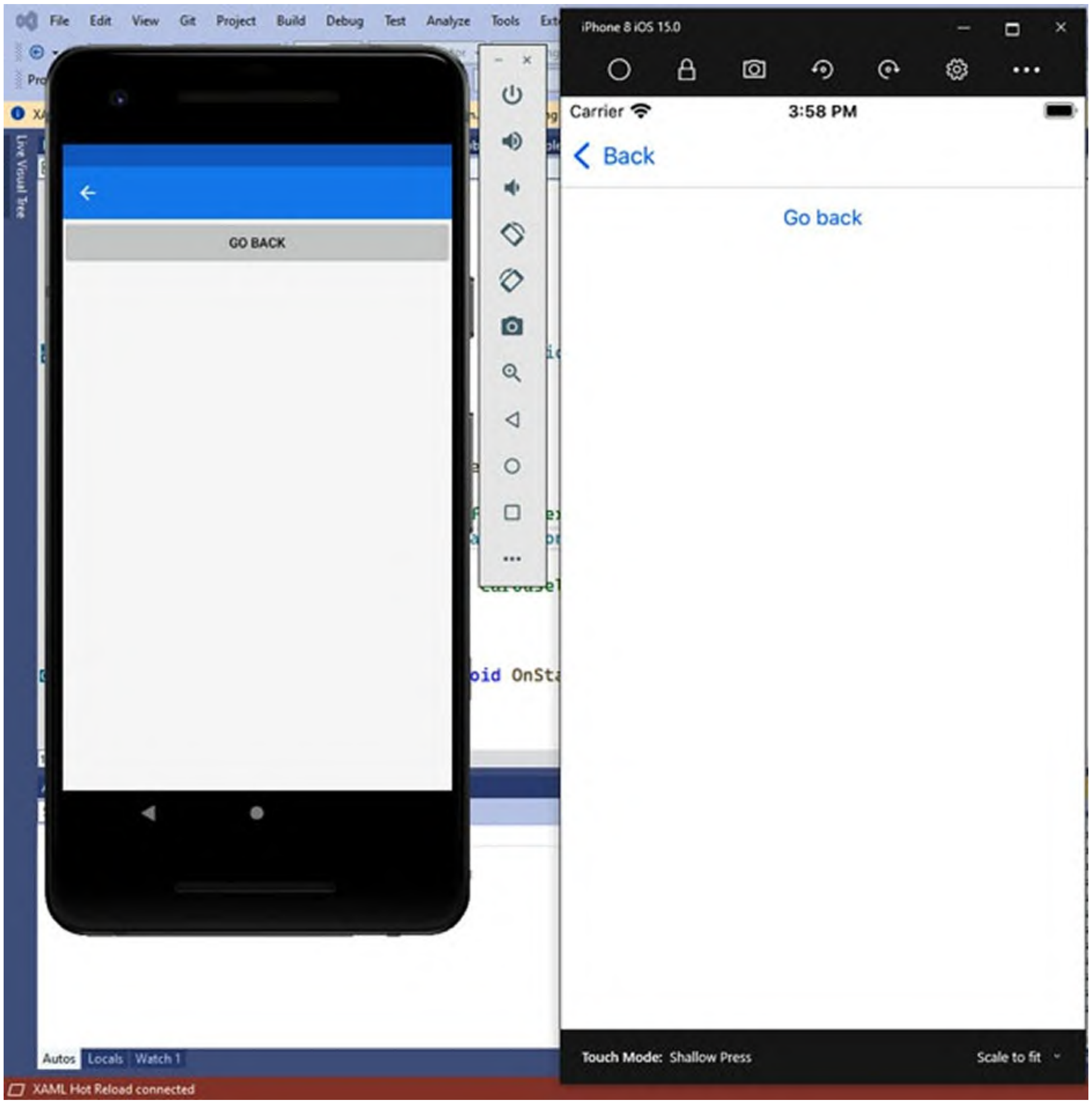


***Figure 8.5:*** *Implementing navigation between pages*

Users can either tap the soft back button at the left side of the navigation bar or, on Android, press the physical back button. The only exception is with modal pages because they do not enable the navigation bar, and therefore, handling the back action is your responsibility.

**Tip: Transition between pages can be animated, which is the default behavior. Both `PushAsync` and `PushModalAsync` have an overload that receives a `bool` parameter that represents whether the transition between pages should be animated. If you pass `false`, the animation**

**will not be applied.**

## Sharing data between pages

Pages often need to exchange objects. You have two options to pass an object from one page to another. The first way is to add an overload of the page's constructor that receives a parameter. The following example shows how to implement a constructor overload that receives an object of type int, which is stored inside a field:

```
public partial class SecondaryPage : ContentPage
{
  public SecondaryPage()
  {
    InitializeComponent();
  }
  private int receivedNumber;
  public SecondaryPage(int oneNumber)
  {
    InitializeComponent();
    receivedNumber = oneNumber;
  }
}
```

With this approach, you can call the page either passing a number or nothing. In case you implement a constructor overload to receive data, remember to always add an invocation to **`InitializeComponent`**, and this must precede any other code. The second option is to just add one constructor that receives data, which means the preceding code would become as follows:

```
public partial class SecondaryPage : ContentPage
{
  private int receivedNumber;
  public SecondaryPage(int oneNumber)
  {
    InitializeComponent();
    receivedNumber = oneNumber;
  }
}
```

This approach is good if the target page must necessarily receive data. For both options, you pass data in the **`PushAsync`** (or **`PushModalAsync`**) invocation, as follows:

```
await Navigation.PushAsync(new SecondaryPage(1000));
```

In this line of code, 1000 is an integer, which is also the type required by the page constructor. In your scenarios, you will pass the appropriate object based on your constructor implementation.

## Implementing custom titles

When you assign the Title property of a page that is wrapped inside a **`NavigationPage`**, the navigation bar will also display the title text. You can customize the title to use a view instead of displaying text by adding a **`NavigationPage.TitleView`** attached property to a page. The following code shows an example:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="IntroducingPages.SecondaryPage">
  <NavigationPage.TitleView>
   <StackLayout Orientation="Horizontal">
    <Image Source="icon.png"/>
    <Label Text="Page title" />
   </StackLayout>
  </NavigationPage.TitleView>
</ContentPage>
```

In the example, the title is made of a layout that contains an image and some text.

## Understanding pages lifecycle

The lifecycle of a page relies on the following two events:

- **`OnAppearing`**, which is fired just before the page is rendered.
- **`OnDisappearing`**, which is fired when the page is being removed from the navigation stack.

They are exposed by any object deriving from **`Page`**, and they do not need to be declared explicitly, unless you want to implement custom actions in those specific moments. If this is the case, their basic code looks as follows:

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

**Tip: IntelliSense can be of great help here. Just type `override`, and then select one of the two method names and press Tab. The code editor will generate the necessary basic code.**

The flow can be summarized as follows:

- When a page is opened for the first time, the constructor is invoked and **`OnAppearing`** is fired.
- When another page is being opened via navigation, **`OnDisappearing`** is invoked on the current page, but the instance of the current page is neither destroyed nor removed from the stack.
- When the page opened via navigation is closed with the invocation to **`PopAsync`** or **`PopModalAsync`**, **`OnDisappearing`** is invoked, and the page instance is removed from the stack.
- When the navigation goes back to the first page, the constructor is not invoked because the instance was not removed from the stack, but **`OnAppearing`** is fired again.

You can, therefore, take advantage of **`OnAppearing`** to write code that you wish to be executed every time a page is opened, whereas in the constructor, you can write code that will be executed only once. Inside **`OnDisappearing`**, you can write code that will be executed every time the current page is leaving, either because it is being closed or because you are navigating to a different page.

## Intercepting the physical back button

Android devices have a physical **`Back`** button that simplifies the navigation between apps or between features in one app. Unlike the software back button on the navigation bar, the physical Back button suspends the current app by default. However, you might want to allow users to go back to the previous page when they press the physical back button. To accomplish this, you can handle the **`OnBackButtonPressed`** event as follows:

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

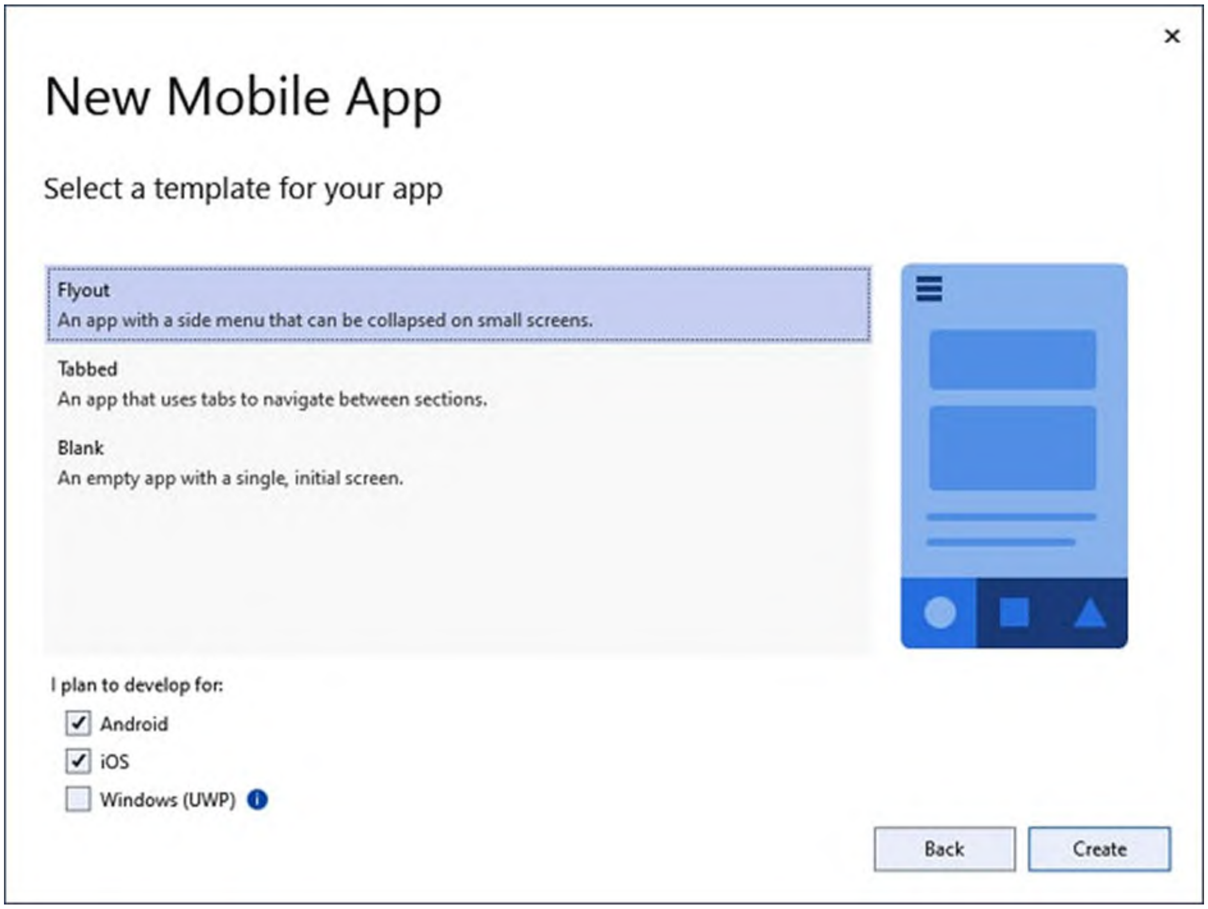
The **`base.OnBackButtonPressed`** method invokes the default behavior, so you might change it as follows:

```
 protected override async bool OnBackButtonPressed()
 {
  await Navigation.PopAsync();
  return true;
 }
```

When you implement your custom logic, the method should return **`true`**, which means the back button was handled. Additionally, note that the method has been marked with the **`async`** operator because it is invoking **`PopAsync`** via the await operator. Handling this event will have no effect on iOS.

## Common app features: The Shell

The Shell is a special root layout that simplifies the way you can implement features that are common to a number of mobile apps, such as a flyout menu, a search bar, navigation features, and a navigation bar. The biggest benefit of the Shell is that all these common features are implemented in one place as part of the Shell itself, instead of doing all the work yourself. The Shell is a very sophisticated layout, so in this chapter, you will learn how to get the most out of it following an approach that is based on real-world experience, which is what you need to know as a Xamarin jobseeker. In the meantime, take a look at *Figure 8.7* and *Figure 8.8*, where you can see a navigation bar and a flyout menu. You will be guided to create such a user interface in this section. Visual Studio offers a project template to create projects based on the Shell, called **Flyout** (see *Figure 8.6*).


New Mobile App

Select a template for your app

Flyout
An app with a side menu that can be collapsed on small screens.

Tabbed
An app that uses tabs to navigate between sections.

Blank
An empty app with a single, initial screen.

I plan to develop for:

- [✓] Android
- [✓] iOS
- [ ] Windows (UWP)

***Figure 8.6:*** *The Flyout project template enables the Shell*

Using this project template eliminates the need to manually add the necessary code to implement the Shell. However, this project template generates code that is already too complex if you have never seen the Shell before, so you are encouraged to open the **`ShellDemo`** companion solution instead, as it allows for implementing features step by step. Assuming that you have opened the companion solution, it is now time to understand how the Shell works.

The Shell is a very articulated object and would require more than one dedicated chapter of a book. This section walks through the Shell with the approach of real-world experience, meaning that you will find all the necessary information you need to work with the Shell in a variety of applications. For more information, you can keep the official documentation as a reference (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/shell**).

## Understanding the structure of the Shell

The Shell is represented by an object called **`AppShell`**, whose declaration can be found in the **`AppShell.xaml`** file. This file will contain all the necessary markup to implement the features discussed in this chapter. The Shell can contain one or more of the following elements:

- **A so-called tab bar**: Also referred to as the navigation bar, it provides buttons with images and text that allow users to navigate between pages. It is automatically placed at the bottom of the Shell (see *Figure 8.7*):

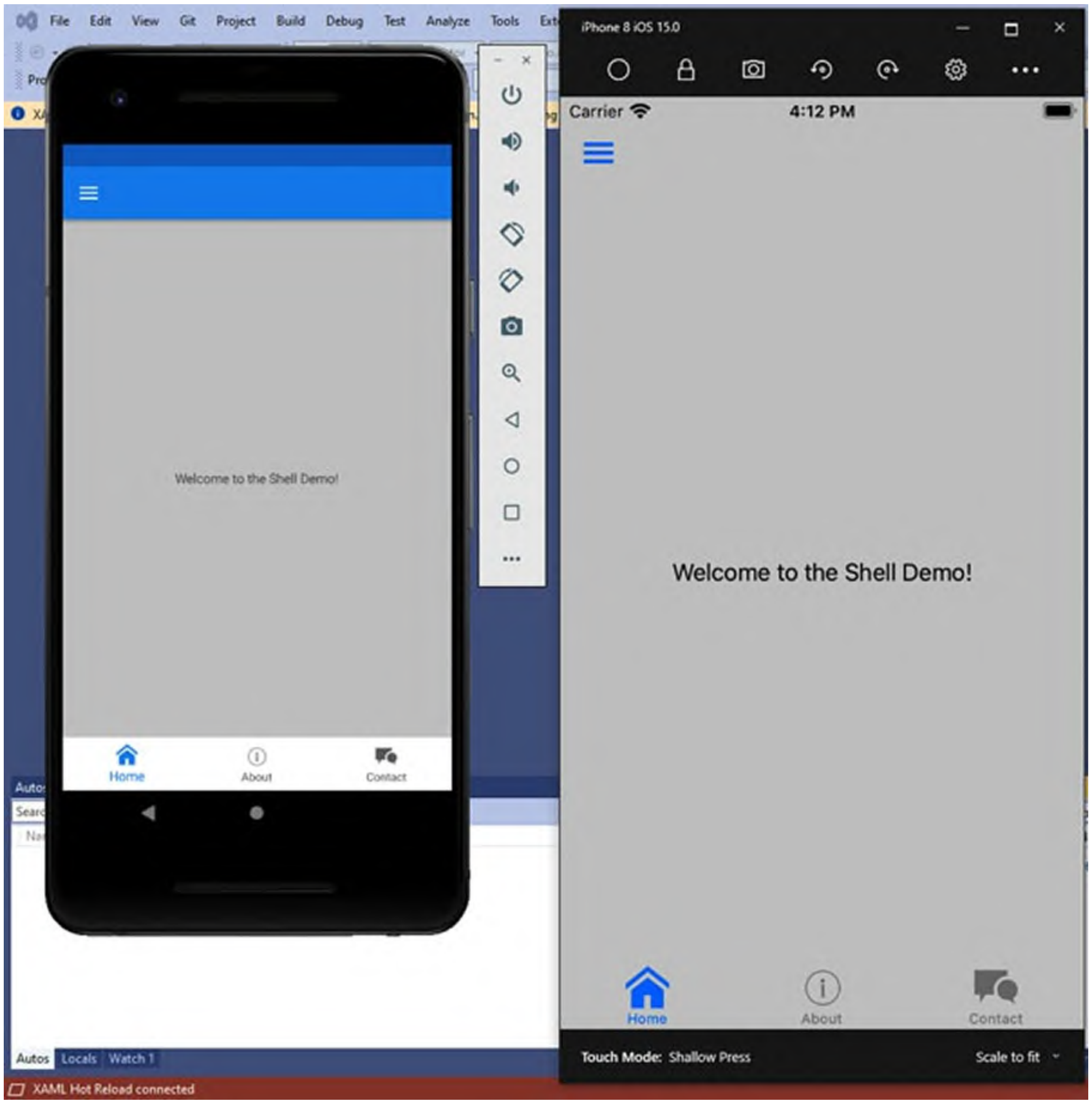


***Figure 8.7:*** *Building apps with the Shell*

- **A flyout menu**: This is similar to what you can get with a **`FlyoutPage`**, but it is all in the same root layout. So, it supports swiping and is normally used to provide shortcuts to other contents. *Figure 8.8* shows an example:

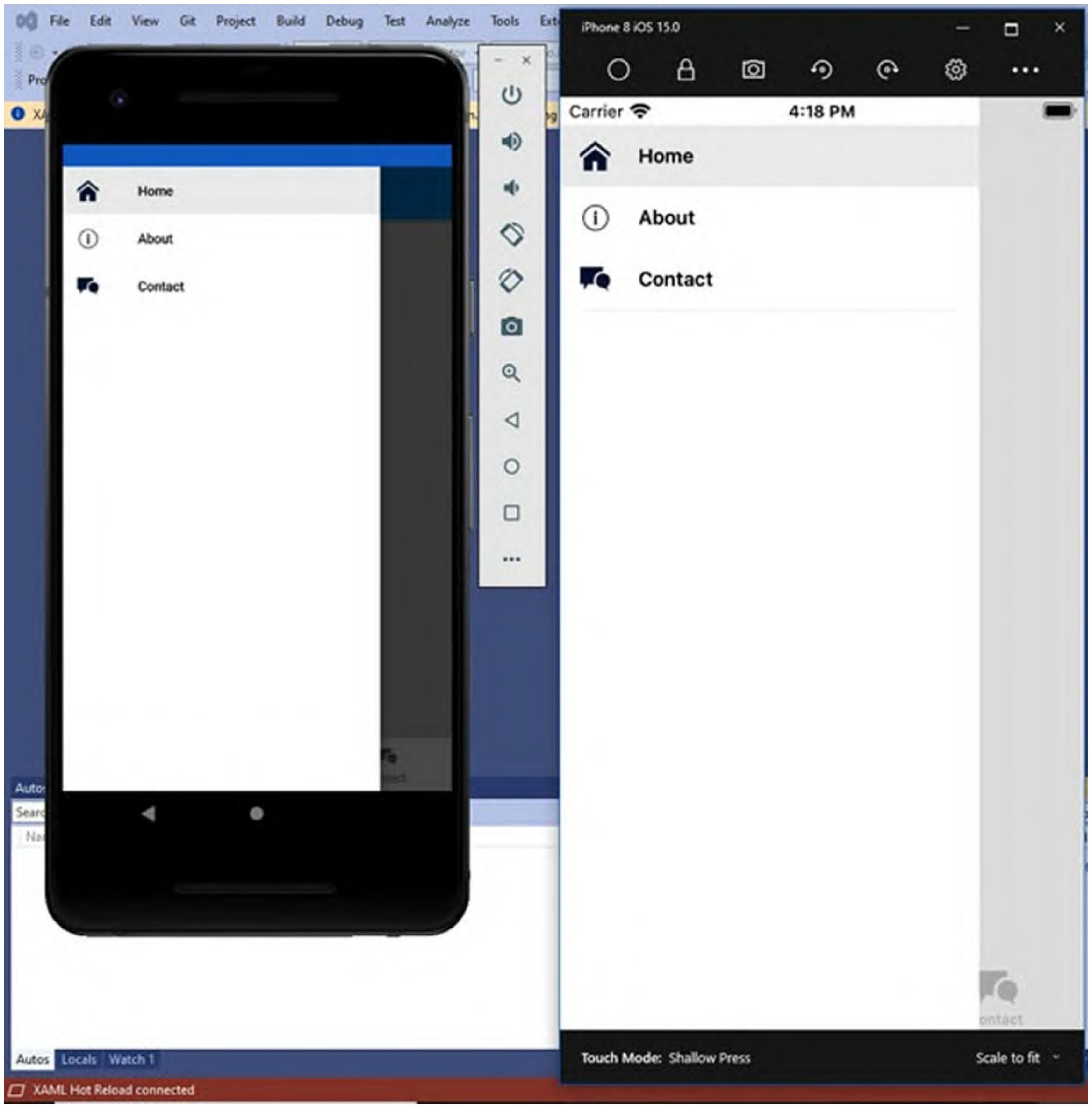


***Figure 8.8:*** *Implementing a flyout*

- **A search bar**: This implements a user interface for searching contents, and you can handle it to display search results.

These visual elements do not have a relationship with one another, so you are not obliged to implement all of them. In addition, they support styles so that you can completely change the appearance of shell items, as you will see in the last section of this chapter. In the upcoming paragraphs, you will learn how to implement all the listed items. You will first learn how to implement a flyout and a tab bar individually, and then you will learn how to implement both with a simplified syntax.

## Adding a Flyout menu

Adding a flyout to the Shell is very easy. In fact, for each item you want to add to the flyout, you declare an object of type **`FlyoutItem`** under the Shell root node. To accomplish this, open the **`AppShell.xaml`** file in Visual Studio. Now, imagine that you want to create a flyout menu with shortcuts to three pages. The following code demonstrates this:

```
<Shell … >
  <FlyoutItem Title="Home" Icon="home.png">
    <ShellContent ContentTemplate="{DataTemplate
    pages:HomePage}"/>
  </FlyoutItem>
  <FlyoutItem Title="About" Icon="about.png">
    <ShellContent ContentTemplate="{DataTemplate
    pages:AboutPage}"/>
  </FlyoutItem>
  <FlyoutItem Title="Contact" Icon="contact.png">
    <ShellContent ContentTemplate="{DataTemplate
    pages:ContactPage}"/>
  </FlyoutItem>
</Shell>
```

The **`Title`** and **`Icon`** properties of each **`FlyoutItem`** object represent the text displayed in the menu and an icon, respectively. Icons follow the same rules as described in the previous chapter about images. The content of each **`FlyoutItem`** is an object of type **`ShellContent`**, which allows for navigating to a different page, but still inside the Shell. The syntax used to specify the target page assigns the **`ContentTemplate`** property with a special object called **`DataTemplate`**. Data templates are one of the topics covered in the next chapter, so for now, just memorize this syntax. *Figure 8.8* shows the result of this code. *Table 8.3* summarizes other interesting properties that you can assign to customize the flyout appearance and behavior.

| Property | Description |
|---|---|
| **`FlyoutIcon`** | Specifies a custom icon file for the flyout menu. |
| **`FlyoutHeader`** | |

| | |
|---|---|
| | Allows for adding a view at the top of the flyout. |
| **`FlyoutBackgroundImage`** | Assigns a background image to the flyout. |
| **`FlyoutBackgroundImageAspect`** | Sets the stretching of the background image. Supported values are `Fill`, `AspectFit`, and `AspectFill` for the `Image` view. |
| **`FlyoutIsPresented`** | Specifies whether or not the flyout should automatically be opened. |

***Table 8.3:*** *Properties to customize the flyout in the Shell*

## Leveraging built-in navigation: The Tab bar

The Shell provides a built-in navigation framework based on tabs. If you look at *Figure 8.7* and *Figure 8.8*, you can see the tab bar at the bottom of the shell, with icons and text. The tab bar is represented by the **`TabBar`** object, and each tab is represented by a **`Tab`** object. Like for the **`TabbedPage`**, there is no limit to the number of tabs you add, but the recommendation is no more than three or four in order to make sure they fit in whatever screen size. The **`TabBar`** is also added to the Shell object, like the Flyout, and for each tab, you add a **`ShellContent`** object, just like you did in the Flyout definition. The following code demonstrates how to create the tab bar you see in *Figure 8.7* and *Figure 8.8*:

```
<Shell … >
  <TabBar>
    <Tab Title="Home" Icon="home.png">
      <ShellContent ContentTemplate="{DataTemplate
      pages:HomePage}"/>
    </Tab>
    <Tab Title="About" Icon="about.png">
      <ShellContent ContentTemplate="{DataTemplate
      pages:AboutPage}"/>
    </Tab>
    <Tab Title="Contact" Icon="contact.png">
      <ShellContent
          ContentTemplate="{DataTemplate pages:ContactPage}"/>
    </Tab>
  </TabBar>
</Shell>
```

The text displayed on each tab is specified via the **`Title`** property. It is worth mentioning that you do not need to write any code to implement page navigation because everything is handled by the Shell (though you have options to do so, which will be covered in the upcoming sections).

## Implementing both the Flyout and Tab bar

Many mobile apps provide the same shortcuts to other pages in both a flyout menu and the tab bar. This allows a better user experience; for example, it allows for accessing a page from the flyout when the shortcut on the tab bar is covered by the flyout itself. If you wish to have the same navigation shortcuts in both a flyout and tab bar, you can write the following code:

```
<Shell … >
  <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
    <Tab Title="Home" Icon="home.png">
      <ShellContent ContentTemplate="{DataTemplate
      pages:HomePage}"/>
    </Tab>
    <Tab Title="About" Icon="library.png">
      <ShellContent ContentTemplate="{DataTemplate
      pages:AboutPage}"/>
    </Tab>
    <Tab Title="Contact" Icon="contact.png">
      <ShellContent
          ContentTemplate="{DataTemplate pages:ContactPage}"/>
    </Tab>
  </FlyoutItem>
</Shell>
```

In summary, you add **`Tab`** objects as children of a **`FlyoutItem`** object, and you must also assign the **`FlyoutDisplayOptions`** property with **`AsMultipleItems`**. This property value makes it possible for one flyout item to display multiple child elements.

## Implementing the Search bar

The Shell also includes a built-in search user interface, which requires some preliminary work. First, you need to create a class that inherits from the base **`SearchHandler`** class, where you implement your own search logic. Suppose you have an application that works with a list of contacts and that you have the following objects to represent people:

```
public class Person
{
  public string FirstName { get;set; }
  public string LastName { get;set; }
}
public class PersonViewModel
{
  public ObservableCollection<Person> People { get; set;}
  public PersonViewModel()
  {
   People = new ObservableCollection<Person>();
   People.Add(new Person { LastName="Del Sole",
   FirstName="Alessandro"});
   People.Add(new Person { LastName = “White", FirstName =
   “Robert" });
   People.Add(new Person { LastName = “Sonny", FirstName =
   “John" });
  }
 }
```

Now, also imagine that the **`PersonViewModel`** class is instantiated somewhere in the app and that its **`People`** property is populated with a list of **`Person`** objects. At this point, you can implement a class the searches the collection for specific items based on a search criterion. In the current example, the search criterion is the person’s last name. The following code demonstrates how to accomplish this:

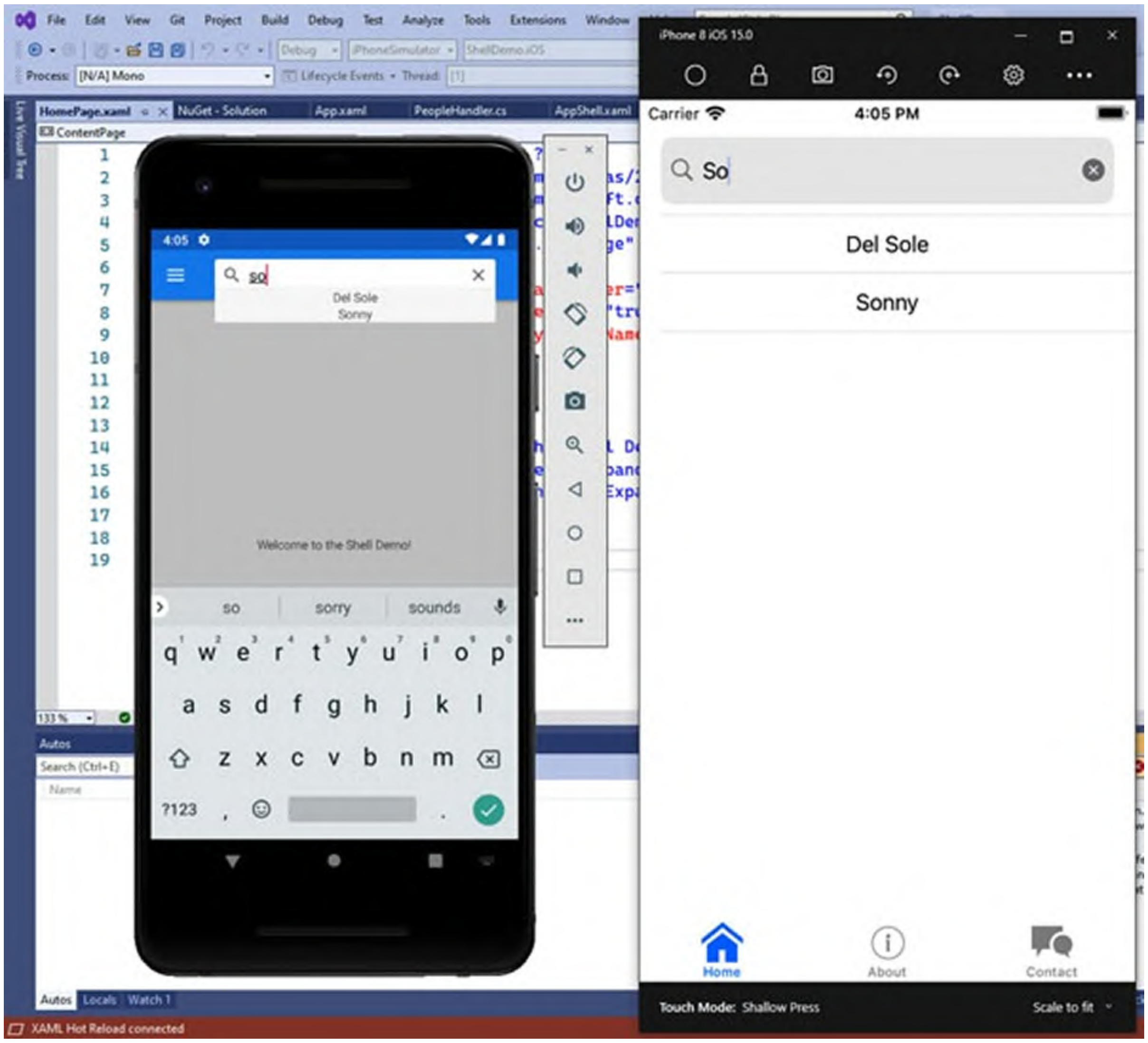
```
 // Requires a using System.Linq directive
 public class PeopleSearchHandler : SearchHandler
 {
  PersonViewModel { get; set; }
  protected override void OnQueryChanged(string oldValue, string
  newValue)
  {
   base.OnQueryChanged(oldValue, newValue);
   if (string.IsNullOrWhiteSpace(newValue))
   {
    ItemsSource = null;
   }
   else
   {
    ItemsSource = PersonViewModel.People
     .Where(p =>
     p.LastName.ToLower().Contains(newValue.ToLower()))
     .ToList();
   }
  }
  protected override async void OnItemSelected(object item)
  {
   base.OnItemSelected(item);
   Person person = item as Person;
   if(person != null)
   {
    await (Application.Current.MainPage as Shell).
     GoToAsync($"PersonDetails?name={person.LastName}");
```

```
      }
    }
  }
```

The **`OnQueryChanged`** event is fired when the user types in the search field. Here, there is a LINQ query that filters the collection of **`People`** object based on what the user typed. The query result is assigned to a property of the search handler class called **`ItemsSource`**. This property can contain a collection of objects, which is displayed under the search bar as you type. Note how **`OnQueryChanged`** also exposes the **`oldValue`** and **`newValue`** strings, representing the previous text in the search bar and the current value in the search bar, respectively. The other event you need to handle is called **`OnItemSelected`**, which is raised when the user selects an item in the search results. Normally, the target action consists of retrieving the instance of the object that was selected from the search results and opening a new page that can process the retrieved data type. In the current example, this target page is called **`PersonDetails`**. When your logic is complete, you can add the search bar to the user interface. Unlike other features, such as the flyout or tab bar, the search bar is not declared in the XAML code of the Shell but at the page level. For example, in the **`HomePage.xaml`** file, you could add the following XAML before the definition of the **`Content`** property:

```
<Shell.SearchHandler>
  <local:PeopleSearchHandler Placeholder="Enter search term"
         ShowsResults="true" SearchBoxVisibility="Expanded"
         DisplayMemberName="LastName" />
</Shell.SearchHandler>
```

*Figure 8.9* shows how the search user interface appears:



Del Sole
Sonny
Welcome to the Shell Demo!
So
Home
About
Contact
Touch Mode: Shallow Press
Scale to fit
XAML Hot Reload connected

***Figure 8.9:*** *The search bar*

When the user selects an item from the list, the corresponding class instance is retrieved, and a new page is opened. It is worth mentioning the **`DisplayMemberName`** property, which specifies the property that is used to display the contents of the search box.

## Programmatically interacting with the Shell

You can interact with the Shell via C# and set its properties as you would do in XAML or if you need to change something at runtime. The Shell is represented by the **`Shell`** class, which is available as a singleton class (one instance only) that you access via the **`Current`** property. For example, the following line of code demonstrates how to programmatically set the background image of the flyout:

```
Shell.Current.FlyoutBackgroundImage =
ImageSource.FromFile(“background.png");
```

You will be able to access all the properties of the objects inside the Shell with the same approach, and IntelliSense will help you understand the available members. The following line of code demonstrates how to programmatically navigate to a different page:

```
await Shell.Current.GoToAsync(“about");
```

The **`GoToAsync`** method allows for opening a different page, but this requires defining a so-called **route**. This can be done by assigning the **`Route`** property of a **`ShellContent`** object with an identifier of your choice, like in the following line:

```
<ShellContent ContentTemplate="{DataTemplate pages:AboutPage}"
Route="about"/>
```

Other objects that support routes are the **`FlyoutItem`**, the **`Tab`** and the **`TabBar`**. You can also invoke **`GoToAsync`** to navigate to a page that is not included in the Shell’s visual hierarchy. To do this, you first register a route for the target page, as follows:

```
Routing.RegisterRoute(“TargetPage", typeof(TargetPage));
```

Where **`TargetPage`** is the name of the destination page that is not included in the visual hierarchy.

**The limited number of C# examples in this section is intentional. Remember, one of the major benefits of XAML is allowing professional designers to work on the user interface without touching any C# code. You should limit your C# interactions with the Shell to the situations highlighted in this section and leave the design changes in XAML.**

## Changing the Shell styles

By default, the Shell takes the theme colors of the target operating system. However, it is possible to customize elements like the flyout and the tab bar with different colors and typefaces. In **`Xamarin.Forms`**, this is done via resources. These are thoroughly discussed in the next chapter, but a preview is offered here. The following code demonstrates how to change colors on the **`Shell`** and the tab bar:

```
<Shell.Resources>
  <ResourceDictionary>
```

```
    <Style x:Key="BaseStyle" TargetType="Element">
      <Setter Property="Shell.BackgroundColor" Value="LightGreen"
      />
      <Setter Property="Shell.ForegroundColor" Value="Blue" />
      <Setter Property="Shell.TitleColor" Value="White" />
      <Setter Property="Shell.DisabledColor" Value="LightGray" />
      <Setter Property="Shell.UnselectedColor" Value="Gray" />
      <Setter Property="Shell.TabBarBackgroundColor"
       Value="LightBlue"/>
      <Setter Property="Shell.TabBarForegroundColor" Value="Red"/>
      <Setter Property="Shell.TabBarUnselectedColor"
       Value="Orange"/>
      <Setter Property="Shell.TabBarTitleColor" Value="Red"/>
    </Style>
    <Style TargetType="ShellContent" BasedOn="{StaticResource
    BaseStyle}"/>
  </ResourceDictionary>
</Shell.Resources>
```

As you can see, property names are self-explanatory. You assign them using a **`Setter`** object for each property, where name and value are represented by **`Property`** and **`Value`**, respectively. They are enclosed inside a **`ResourceDictionary`** resource container, wrapped inside a **`Shell.Resources`** collection. Note how a style is also added for the **`ShellContent`** object, which is the core of the Shell and is based on the previous style. The result of this styling is shown in *Figure 8.10*, which gives you an idea of how the different colors are applied:

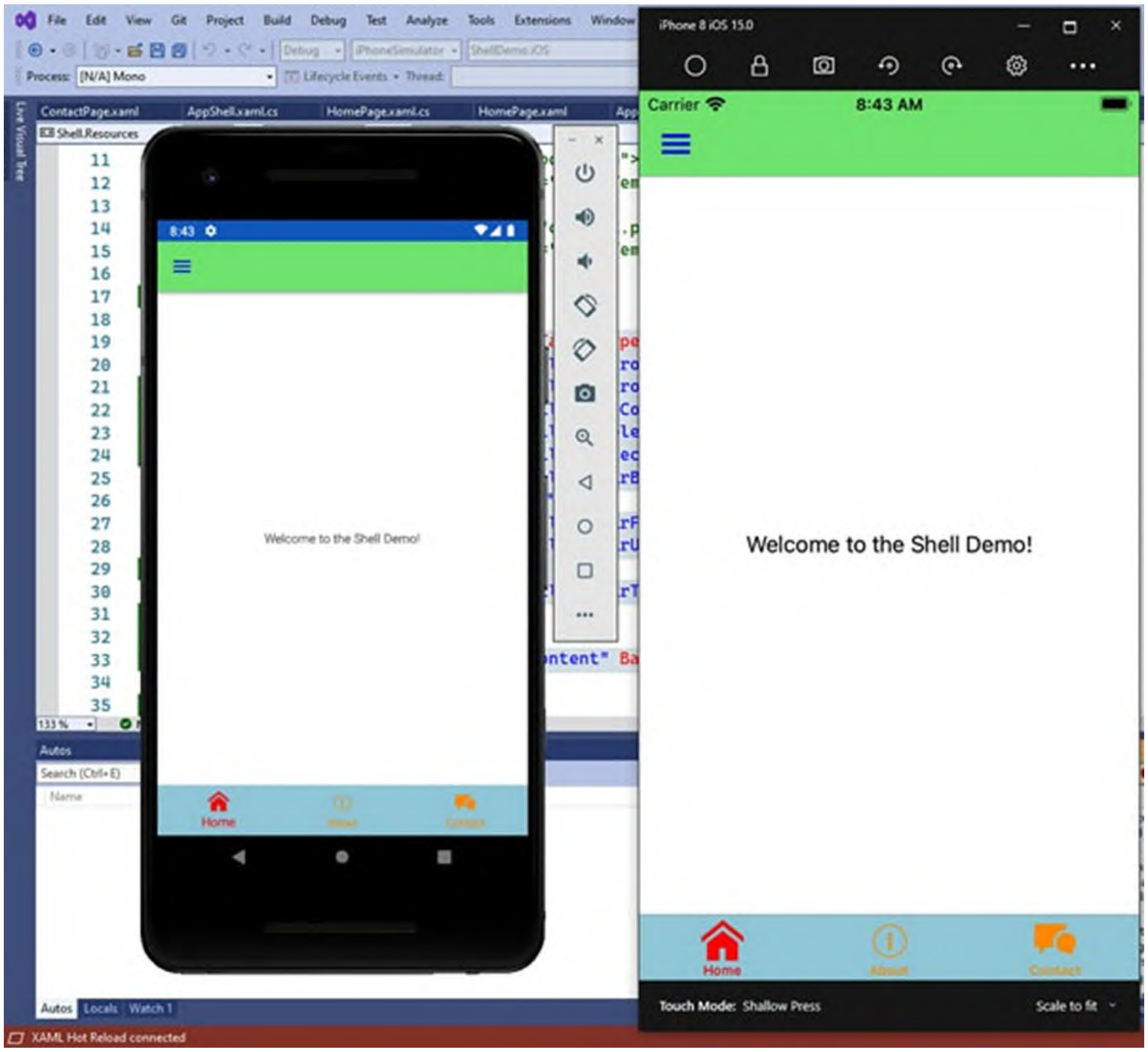

iPhone 8 iOS 15.0
Carrier
8:43 AM
Welcome to the Shell Demo!
Home
About
Contact
Touch Mode: Shallow Press
Scale to fit
Welcome to the Shell Demo!
XAML Hot Reload connected

***Figure 8.10:*** *Restyling the Shell*

## Conclusions

The user experience in mobile apps can be different and more complex, depending on the purpose of the app. In **`Xamarin.Forms`**, you can define sophisticated user interfaces by leveraging different kinds of pages. The **`ContentPage`** represents an individual page; the **`FlyoutPage`** represents a master-details view with a flyout menu; the **`TabbedPage`** groups individual pages within tabs; the **`CarouselPage`** allows for implementing the swipe gesture over a series of individual pages. It is also very common to implement navigation between pages, which is accomplished via the **`NavigationPage`** class and that offers a built-in navigation framework, including handling a back gesture. In addition, it is possible to use the Shell, a convenient feature that makes it simple to create apps with features like flyout, navigation, and search from a single place. Now you have very good knowledge of how to build the user interface in `Xamarin.Forms`, you are ready for the next step: working with data.

## Key terms

- **Page**: A root visual element that contains one view, such as a layout, and whose purpose is offering a specific functionality.
- **Flyout**: A side menu that can be shown and hidden with a swipe gesture.
- **Navigation stack**: A system object that contains the list of open pages and that works with a **Last-in, First-out** (**LIFO**) approach.
- **Shell**: A root container that implements common app features in one place to simplify the architecture of an app.

## Suggested readings

- The official Xamarin.Forms documentation about pages and navigation: **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/navigation**
- The official Xamarin.Forms documentation about the Shell: **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/shell**

# CHAPTER 9
# Resources and Data Binding

## Introduction

In real-world mobile app development, you will often need to follow a graphic design that forces you to repeat colors, fonts, and size over the same type of views. Instead of assigning the same properties to each view every time, you can leverage resources. The syntax for using resources is similar to the one you use to leverage another powerful feature called data binding, which allows connecting views to .NET objects so that they automatically exchange information without your manual intervention. For example, selecting a date from a `DatePicker` and storing the selection result into a `DateTime` variable can be an automated process, and this approach becomes even more important when you work with data collections and databases. This chapter describes both resources and data binding, providing you with a crucial part of the knowledge about Xamarin.Forms.

**Data access, data binding and topics-related views you use for these scenarios can be very complex. Therefore, the goal of this chapter is not to describe all the available properties and possible uses of all the objects. Instead, as a Xamarin jobseeker, the goal is to give you all the necessary knowledge you need to immediately be productive on existing projects for both maintenance and implementation of new features.**

## Structure

In this chapter, we will cover the following topics:

- Understanding and defining resources
- Binding data to the user interface
- Advanced data binding: The Model-View-ViewModel pattern
- Local data access with SQLite databases

## Objectives

After completing this chapter, you will be able to define and reuse styles and data templates, and you will be able to automate the communication between the user interface and data collections, passing through advanced programming patterns and local data access with the SQLite database.

## Creating a sample project

This chapter comes with three different Xamarin.Forms solutions that you can open with Visual Studio to better follow the examples. If you wish to recreate the projects on your own, you can follow these steps:

1. Create three new Xamarin.Forms solutions called **`Resources`**, **`DataBinding`** and **`MvvmSample`**, respectively. For the local data access part, you will extend the **`MvvmSample`** project.
2. For each shared project, do not edit or remove the auto generated **`MainPage.xaml`** file as it will be used later.
3. For each view discussed in the book, especially about data binding, add a new item of type **`Content Page (XAML)`**. To accomplish this, right-click on the shared project name and then click on **`Add New Item`** in the **`context`** menu.
4. In the **`Add New Item`** dialog, click on the Xamarin.Forms node on the left and then select the **`Content Page (XAML)`** item template.
5. Assign a name that matches the discussed view, for example **`CollectionViewSample.xaml`**, to the new XAML file and click on **`Add`**.

For each page you add to the project, add an empty **`StackLayout`** to the **`ContentPage`** and assign its **`VerticalOptions`** property with **`CenterAndExpand`**. Unless where specified, this will be the layout of choice for the code examples in the next pages.

## Understanding and Defining Resources

In software development, resources represent reusable objects. In .NET development, resources are typically represented by **`.resx`** files where you can define reusable strings or images. Instead, in Xamarin.Forms, and more generally in all XAML-based development platforms like WPF and UWP, resources represent reusable sets of properties defined in XAML that you can apply to multiple visual elements. Xamarin.Forms supports several types of resources: styles, data templates and object references. Styles will be discussed in the upcoming paragraphs, whereas data-templates and object references will be discussed when talking about data binding because this is where they are used. But first, you need to know how to declare resources and their visibility scope.

## Defining resources

The **`Application`** class, pages and layout all expose a collection called **`Resources`**. This is a collection of **`ResourceDictionary`** objects, where each can contain different resources. In practice, you do not really need to define multiple **`ResourceDictionary`** instances, but this can be useful if you want to group resources logically. You can define a collection of resources as follows:

```
<?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="Resources.Page1">
  < ContentPage.Resources>
   <ResourceDictionary>
   </ResourceDictionary>
  </ ContentPage.Resources>
</ ContentPage >
```

In this code snippet, resources are being declared at the page level. In the following example, resources are being defined at the layout level:

```
<Grid>
  <Grid.Resources>
   <ResourceDictionary>
   </ResourceDictionary>
  </Grid.Resources>
</Grid>
In the following code snippet, resources are defined at the app
level:
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Resources.App">
  <Application.Resources>
   <ResourceDictionary>
   </ResourceDictionary>
  </Application.Resources>
```

```
</Application>
```

The reason why these three examples have been provided is the resources' scope, which can be summarized as follows:

- Resources defined in the **`App.xaml`** file are available across the app.
- Resources defined in a page are only available to views and layouts included in the page itself and not outside.
- Resources defined in a layout are only available to views and layouts included in the layout itself and not outside.

Now that you know where resources are defined and how their scope works, you are ready to start defining resources in practice.

## Defining and assigning styles

Styles are the most common type of reusable resource and the easiest way to get started with this topic. A style can be thought of as a reusable set of properties and values that can be applied to visual elements of the same type. For example, you might need to apply the same background color, font size, and text color to all the buttons on a page. Another example could be applying the same typeface to all the labels in your app. Without styles, you would need to assign the same properties to all the desired views manually, multiple times. A style is represented by the **`Style`** object. It requires an **`x:Key`** tag that contains a unique identifier for the style and that you will use to assign the style; the key can be thought of as a name. It also requires specifying the view to which the style can be applied, and this is specified via the **`TargetType`** property of the style. Each property you want to be part of the style is represented by the **`Setter`** object, which has two properties: **`Property`**, which specifies the target property, and **`Value`**, which represents the property value. The following code demonstrates how to define a style that targets the **`Button`** view and assigns the **`BackgroundColor`**, **`TextColor`**, **`FontAttributes`** and **`FontSize`** properties:

```
<?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="Resources.App">
  <Application.Resources>
   <ResourceDictionary>
    <Style x:Key="PrimaryButtonStyle" TargetType="Button">
```

```
      <Setter Property="BackgroundColor" Value="Blue"/>
      <Setter Property="TextColor" Value="White"/>
      <Setter Property="FontAttributes" Value="Bold"/>
      <Setter Property="FontSize" Value="Medium" />
    </Style>
   </ResourceDictionary>
  </Application.Resources>
</Application>
```

Consuming a style is then very easy. Views expose a **`Style`** property that you can assign with a style that targets the view of interest. The following code snippet demonstrates how to assign the style defined above to three buttons so that they can all have the same look:

```
<StackLayout VerticalOptions="CenterAndExpand"
HorizontalOptions="FillAndExpand"
    Spacing="20" Margin="20,0,20,0">
  <Button Text="Button one" Style="{StaticResource
  PrimaryButtonStyle}" />
  <Button Text="Button two" Style="{StaticResource
  PrimaryButtonStyle}" />
  <Button Text="Button three" Style="{StaticResource
  PrimaryButtonStyle}" />
</StackLayout>
```

The syntax for assigning the style requires a markup extension enclosed between brackets, made of the **`StaticResource`** object, followed by the key to the style.

**Tip: As you can see as you type, IntelliSense will not only show a list of styles to make you write code faster, but, in particular, it will show only those styles that can be applied to the view you are editing.**

Actually, there is another way to assign a style, which is as follows:

```
<Button Text="Button one" Style="{DynamicResource ButtonStyle}"
/>
```

You can use **`DynamicResource`** instead of **`StaticResource`** if you plan to modify the style's properties at runtime and want to make the target views automatically update to reflect changes. *Figure 9.1* shows the result of the assignment of the style defined previously:

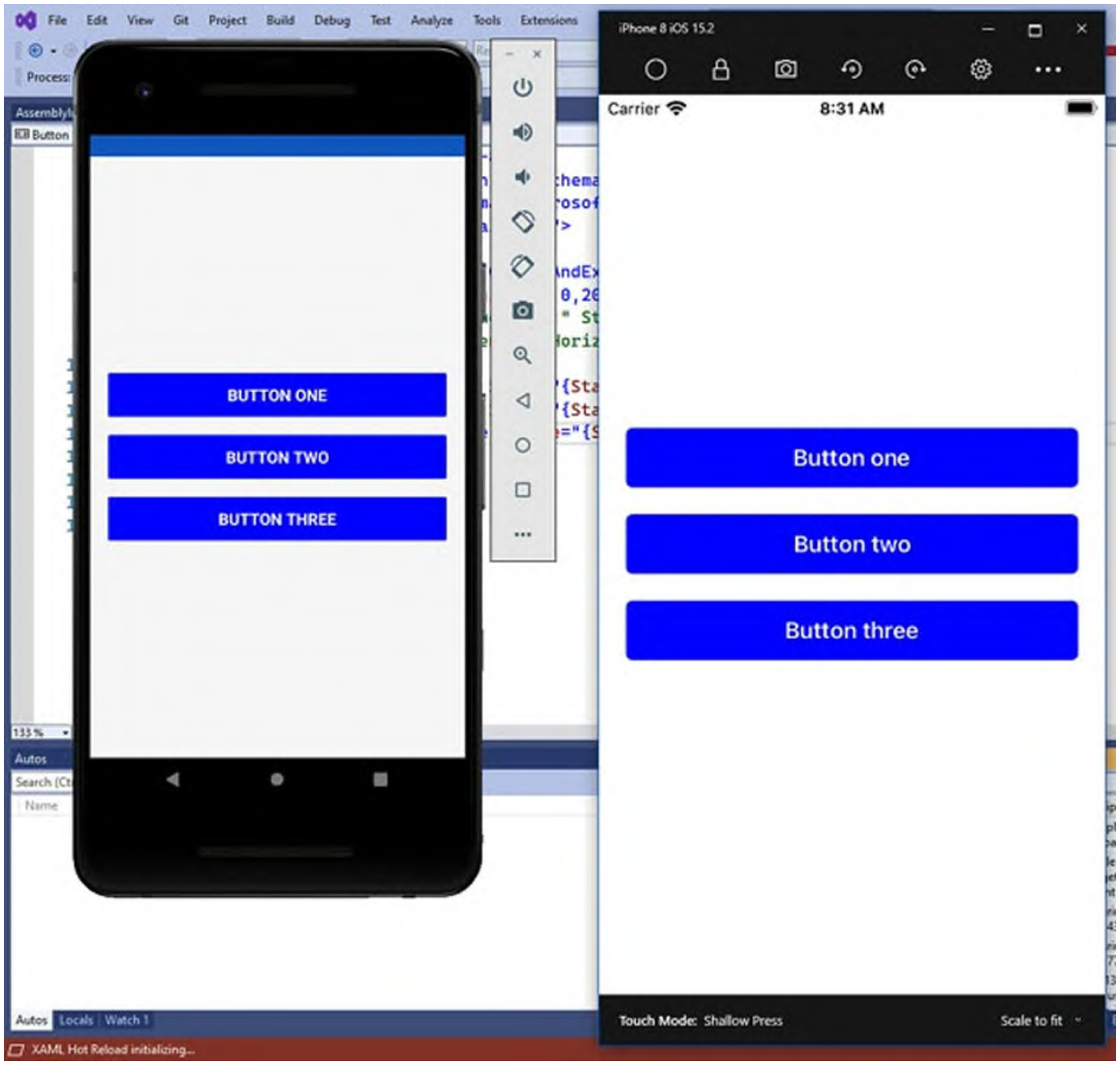


***Figure 9.1:*** *Assigning a style to three different buttons*

Obviously, you can have multiple styles that target the same view and then choose the most appropriate one depending on the page definition and context. The following code demonstrates how to implement multiple styles for the **`Button`** view and how they can be differentiated by specifying a different **`x:Key`**:

```
<Style x:Key="PrimaryButtonStyle" TargetType="Button">
  <Setter Property="BackgroundColor" Value="Blue"/>
  <Setter Property="TextColor" Value="White"/>
  <Setter Property="FontAttributes" Value="Bold"/>
  <Setter Property="FontSize" Value="Medium" />
</Style>
<Style x:Key="SecondaryButtonStyle" TargetType="Button">
```

```
  <Setter Property="BackgroundColor" Value="LightGray"/>
  <Setter Property="TextColor" Value="Blue"/>
  <Setter Property="FontAttributes" Value="Bold"/>
  <Setter Property="FontSize" Value="Medium" />
</Style>
```

## Implementing style inheritance

Styles support inheritance makes it possible to define a base style and then customize the style according to your specific needs. A style can be inherited from another one by specifying the **`BaseOn`** property, which takes a **`StaticResource`** markup extension as the value. The following code demonstrates how to define a style for all the objects deriving from **`View`**, such as views and layouts, where the alignment options are provided. Next, a style for the **`Label`** view is defined and inherits from the previous one. This new style will have the properties of the **`ViewStyle`** style, plus the new ones defined in the **`LabelStyle`** style:

```
<Style x:Key="ViewStyle" TargetType="View">
  <Setter Property="HorizontalOptions" Value="Center" />
  <Setter Property="VerticalOptions" Value="Center" />
</Style>
<Style x:Key="LabelStyle" TargetType="Label"
   BasedOn="{StaticResource ViewStyle}">
  <Setter Property="TextColor" Value="Green" />
  <Setter Property="FontSize" Value="Large" />
</Style>
<Style x:Key="RedLabelStyle" TargetType="Label"
   BasedOn="{StaticResource LabelStyle}">
  <Setter Property="TextColor" Value="Red" />
</Style>
```

The third style, **`RedLabelStyle`**, demonstrates how to redefine one property of the style it inherits from. This will allow the style to have all the properties defined in the base style, plus the changes implemented here.

## Implementing implicit styling

Suppose you want to apply the same style to all the target views defined in the resources' scope. It is possible to leverage a feature called implicit styling. Consider the following code:

```
<Style TargetType="Button">
  <Setter Property="BackgroundColor" Value="Blue"/>
  <Setter Property="TextColor" Value="White"/>
  <Setter Property="FontAttributes" Value="Bold"/>
  <Setter Property="FontSize" Value="Medium" />
</Style>
```

As you can see, there is no **`x:Key`** tag specified. If this style is defined in the application resources, it will be automatically applied to all buttons in the application, without the need to explicitly assign the **`Style`** property. Similarly, if it is defined in the page's resources, it will be applied to all buttons in the page, and if it is defined in a layout's resources, it will be applied to all buttons inside the layout. You can still define additional styles by specifying an **`x:Key`** tag and manually assigning them to the individual view that you do not want to be implicitly styled.

## Binding data to the user interface

In software development, data binding is a mechanism that connects a view to a data object for implementing automatic information exchange between the two. This makes it possible for the view to automatically update its content when the data changes, and vice versa. For better understanding, consider the scenario where the user enters some text into an **`Entry`** and the input string should be stored inside the property of a class. Without data binding, what you should manually do is handle the **`TextChanged`** event for every keystroke and store the string into the data property. What you would also need to do is write code that updates the content of the **`Entry`** when the value of the data property changes at runtime, for example when data is loaded from a database. If this approach does not seem problematic with one entry, it really is with data collections and, in general, with amount of data that you do not know in advance (except for the structure of the class representing the data). Data binding brilliantly solves these problems, implementing automatic communication and content updates between views and other .NET objects (including other views). In the following paragraphs, you will learn how to successfully work with data binding in Xamarin.Forms, leveraging all the power of XAML.

## Getting started with data binding

Because of the complexity and importance of data binding, you will learn this feature in a step-by-step approach, starting with small pieces. Suppose you have the following **`Contact`** class:

```
public class Contact
```

```
{
  public string LastName { get; set; }
  public string FirstName { get; set; }
  public DateTime DateOfBirth { get; set; }
  public bool IsFamilyMember { get; set; }
}
```

Imagine that you have an instance of this class and that you want users to enter data into the class instance via appropriate views in the user interface. You could have a page with two **`Entry`** views, a **`DatePicker`** and a **`CheckBox`** (**`SimpleBindingPage.xaml`** in the sample solution). Instead of manually handling events and data storage for each property, you can use data binding. The XAML code for the page could be the following:

```
<?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="DataBinding.SimpleBindingPage">
  <ContentPage.Content>
    <StackLayout Orientation="Vertical" Padding="20">
      <Label Text="First Name:" />
      <Entry Text="{Binding FirstName}"/>
      <Label Text="Last Name:" />
      <Entry Text="{Binding LastName}"/>
      <Label Text="Date of birth:"/>
      <DatePicker Date="{Binding DateOfBirth, Mode=TwoWay}"/>
      <StackLayout Orientation="Horizontal">
        <Label Text="Is family member?:"
          VerticalOptions="Center"/>
        <CheckBox IsChecked="{Binding IsFamilyMember}"
          VerticalOptions="Center"/>
      </StackLayout>
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

As you can see, for each view, the property that receives the user input (**`Entry.Text`**, **`DatePicker.Date`**, and **`CheckBox.IsChecked`**) is assigned with a markup extension represented by the binding literal plus the target property in the data object.

**Tip: The full syntax for binding to a property would be `{Binding Path=PropertyName}`, but in simple expressions like the previous ones, `Path` can be omitted.**

Xamarin.Forms supports the five types of data binding described in *Table*

9.1, which can be specified via the **`Mode`** property of the binding expression:

| Type | Description |
|---|---|
| **`TwoWay`** | Data binding works in read-write mode (views can read from and write to the bound object). |
| **`OneWay`** | Data binding works in read-only mode, and views can only read from the bound object. |
| **`OneWayToSource`** | Data binding works in write-only mode, and views can only write to the bound object. |
| **`OneTime`** | Data binding works in read-only mode, but views can only read data once. |
| **`Default`** | The most appropriate type is automatically assigned by Xamarin, depending on the current view. This is also the default behavior if no type is specified. |

***Table 9.1:*** *Supported types of data binding*

If you look at the previous code, you can see how no mode is specified for the **`Entry`** views. This assumes **`Mode=Default`**, and Xamarin automatically resolves **`TwoWay`** as the most appropriate one for this kind of view. The **`DatePicker`** is an exception, because it requires specifying the **`TwoWay`** mode even if it natively supports read-write binding. All the properties of a view cannot be data-bound to another property, only the so-called bindable properties can be.

**In this book, you will learn how to consume bindable properties, but their implementation is left to you for further studies. Reasons for this are their complexity and the fact that, as a Xamarin jobseeker, you will realize that bindable properties are commonly only implemented in custom views. The official documentation offers further guidance and is available at https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/bindable-properties.**

The power of bindable properties is the fact that they cannot only write their value into the target object, but they are also automatically updated if the target object's value is updated outside of the data binding. To clarify this, consider a scenario where data is read from a database and then stored inside .NET objects. Data is read, the value is written into properties of a .NET class, and properties of views that are data-bound to such a class automatically update their value. However, bindable properties can automatically refresh their content only if the bound .NET object sends a so-called **property changed notification**, which means informing the user interface that one or more of its values have changed. To accomplish this, the bound class must implement the **`System.ComponentModel.INotifyPropertyChanged`** interface.

## Property change notifications: INotifyPropertyChanged

Bindable properties, which are the source of the data binding, continuously listen for an event called **`PropertyChanged`**, which is raised by objects that implement the **`INotifyPropertyChanged`** interface. When such an event is raised, the bindable property value is automatically refreshed. With this in mind, the **`Contact`** class shown previously should be rewritten as follows:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace DataBinding
{
 public class Contact : INotifyPropertyChanged
 {
  private string _lastName;
  public string LastName
  {
   get
   {
    return _lastName;
   }
   set
   {
    _lastName = value;
    OnPropertyChanged();
   }
  }
  private string _firstName;
  public string FirstName
  {
   get
   {
    return _firstName;
   }
   set
   {
    _firstName = value;
```

```
        OnPropertyChanged();
      }
    }
    private DateTime dateOfBirth;
    public DateTime DateOfBirth
    {
      get
      {
        return dateOfBirth;
      }
      set
      {
        dateOfBirth = value;
        OnPropertyChanged();
      }
    }
    private bool _isFamilyMember;
    public bool IsFamilyMember
    {
      get
      {
        return _isFamilyMember;
      }
      set
      {
        _isFamilyMember = value;
        OnPropertyChanged();
      }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged([CallerMemberName]
        string propertyName = null)
    {
      PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

The following is a list of key points about the code above:

- The **`set`** methods of each property invoke a method called **`OnPropertyChanged`** every time their value changes. This enables for sending a change notification.
- The **`OnPropertyChanged`** method has been defined in order to simplify the invocation for change notification with a single method call.

- The **`CallerMemberName`** attribute automatically resolves the name of the caller member, in this case, the property name that is calling the method. This eliminates the need to specify the property name in the method invocation every time.
- Several objects in the Xamarin.Forms libraries expose a method called **`OnPropertyChanged`** with the same implementation, so doing the same thing in your objects helps you ensure consistency.

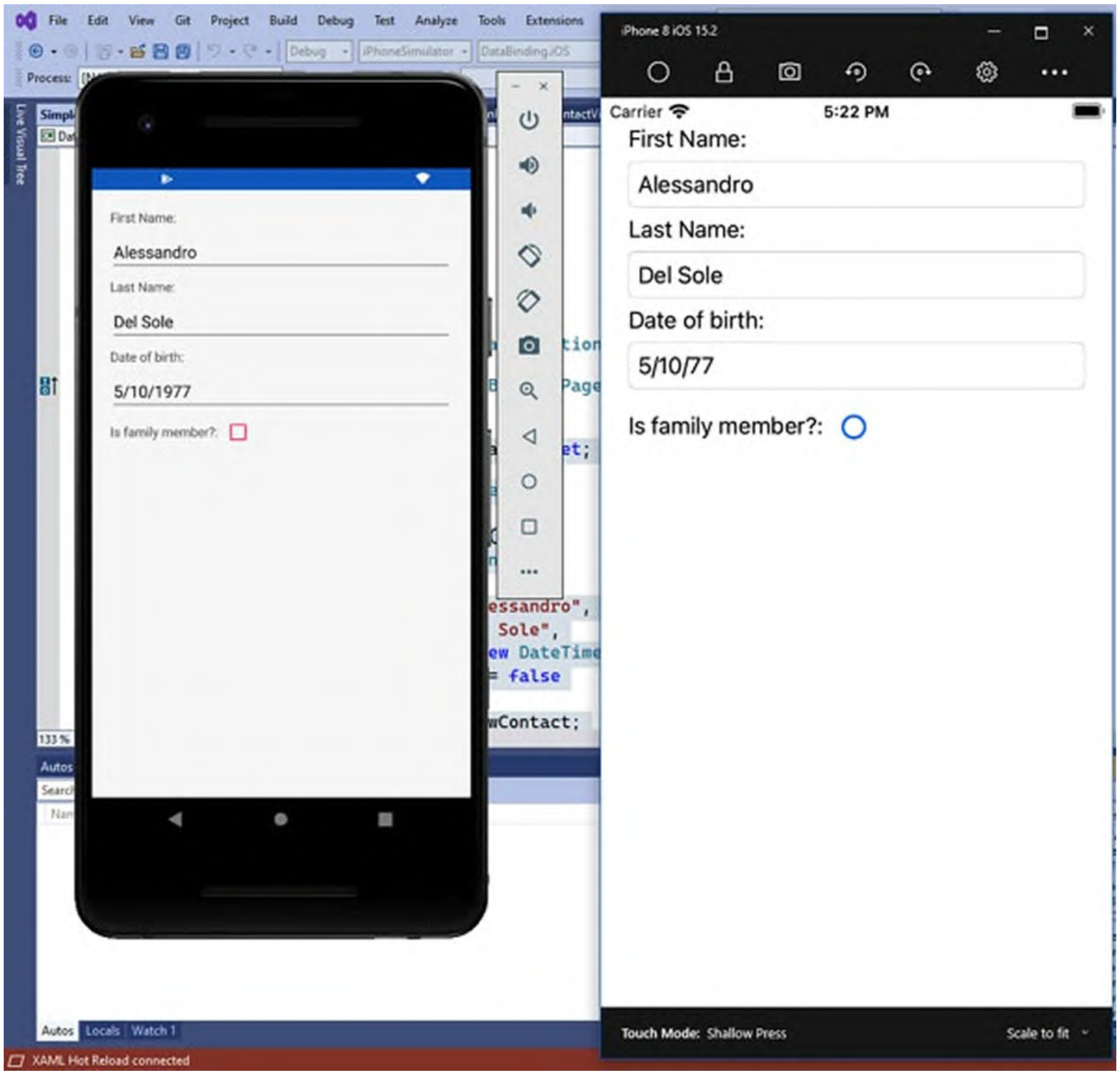
The next step is connecting the object to views and making the data binding alive.

## Assigning the binding context

You have previously created a data object, written some user interface, and prepared views for data binding; the last step is to establish a connection between views and objects. This is accomplished by assigning a property called **`BindingContext`** of the object type. This property is available in many objects, for example, pages and layouts, and it represents the data source for the view. With regard to the previous example, you could write the following code in the page’s code-behind file:

```
private Contact NewContact { get; set; }
public SimpleBindingPage()
{
  InitializeComponent();
  NewContact = new Contact
  {
   FirstName = “Alessandro",
   LastName = “Del Sole",
   DateOfBirth = new DateTime(1977, 05, 10),
   IsFamilyMember = false
  };
  BindingContext = NewContact;
}
```

A new instance of the **`Contact`** class is created and assigned to a property called **`NewContact`** of the **`Contact`** type. The approach of defining a property is recommended because the data binding engine is optimized for working with properties rather than with fields or local variables, and it also makes the object accessible from elsewhere in the page. The **`NewContact`** instance is assigned to the **`BindingContext`** property of the page. This is what really enables data binding. Behind the scenes, child views will look for an object instance inside the **`BindingContext`** property of their parent container and will bind to the properties specified in the binding expressions. If you now run the sample code, you will see how the user interface of the page is automatically filled in with values from the **`Contact`** class instance, as shown in *figure 9.2*:



First Name:
Alessandro
Last Name:
Del Sole
Date of birth:
5/10/1977
Is family member?:

Carrier 5:22 PM
First Name:
Alessandro
Last Name:
Del Sole
Date of birth:
5/10/77
Is family member?:

***Figure 9.2:*** *Binding an object instance to views*

Any changes that you make to the data will be sent to the bound object. You can easily demonstrate this by placing a breakpoint on the **`OnPropertyChanged`** method definition in the **`Contact`** class. You will see how this is invoked every time a property value changes depending on the user input. What you have seen so far is a basic implementation of data binding, which is useful with individual pages, for example, in data-entry scenarios. However, the power of data binding is unleashed when you work with data collections.

## Working with data collections

Displaying and editing data collections in the user interface is another common scenario when it comes to data binding. Xamarin.Forms provides several views that allow for displaying data collections, as summarized in *Table 9.2*:

| View | Description |
|---|---|
| `ListView` | Displays a list of items from a bound collection. |
| `CollectionView` | Displays a list of items from a bound collection with a more modern and efficient approach. |
| `CarouselView` | Allows for scrolling through a list of items from a collection, keeping the focus on the selected item. |
| `TableView` | Displays items from a collection in a tabular view. |
| `Picker` | Allows for selecting one item from a collection, based on the system user interface. |

***Table 9.2:*** *Views that support data collections*

Two important considerations need to be made about the views listed in *Table 9.2*:

- The **`ListView`** was the only view available to display and edit lists until Xamarin.Forms 4.5, which introduced the **`CollectionView`**. Though less efficient and more complex to manage, in the real-world you will still find tons of projects relying on the **`ListView`**. This is why it will be discussed in detail, like any other view.
- Using the **`TableView`** makes sense when you want to implement a series of settings or options. More sophisticated results can be achieved with the **`CollectionView`**, so you will be introduced to the **`TableView`** without going into unnecessary details.

All the views described in *Table 9.2* share one characteristic: they all expose a property called **`ItemsSource`**, of the object type, which receives an instance of the collection you want to bind. The collection must implement the **`IEnumerable`** interface. With Xamarin.Forms, you typically use collections of the **`System.Collections.ObjectModel.ObservableCollection<T>`** type. This special collection works exactly like a **`List<T>`**, but it also sends a change notification when items are added or removed from the collection so that bound views can automatically refresh their content.

**As the `ObservableCollection` sends change notification when items are added or removed, if you want to notify the user interface of changes over an existing item in the collection, you must ensure that the data object implements the `INotifyPropertyChanged` interface discussed previously.**

## Displaying data: The ListView

For years, the **`ListView`** has been the only available view to display data collections, so it is important to understand how it works, especially if you will need to work on maintaining existing projects in the future. If you want to follow the companion sample solution, you can open the **`ListViewSample.xaml`** page. Suppose you have a collection of **`Contact`** objects and you want to display the property values of each contact in a list. The following code demonstrates how to declare an appropriate **`ListView`**:

```
<?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="DataBinding.ListViewSample">
  <ContentPage.Content>
    <StackLayout>
      <ListView x:Name="ContactList" ItemsSource="{Binding}"
         ItemSelected="ContactList_ItemSelected"
         HasUnevenRows="True">
        <ListView.ItemTemplate>
          <DataTemplate>
            <ViewCell.View>
              <Frame HasShadow="False" BorderColor="Blue"
                 Margin="10">
                <StackLayout>
                  <Label Text="Last name:"/>
                  <Entry Text="{Binding LastName}"/>
                  <Label Text="First name:"/>
                  <Entry Text="{Binding FirstName}"/>
                  <Label Text="Date of birth:"/>
                  <DatePicker Date="{Binding DateOfBirth,
                  Mode=TwoWay}"/>
                  <StackLayout Orientation="Horizontal">
                  <Label Text="Is Family Member?:"
                  VerticalOptions="Center"/>
                  <CheckBox IsChecked="{Binding
                   IsFamilyMember}"
                  VerticalOptions="Center"/>
                  </StackLayout>
                </StackLayout>
              </Frame>
            </ViewCell.View>
            </DataTemplate>
        </ListView.ItemTemplate>
       </ListView>
    </StackLayout>
  </ContentPage.Content>
 </ContentPage>
```

There are several key points in this code, and some of them are also shared across views, i.e., they are not limited to the **`ListView`**. The first consideration is the **`ItemsSource`** property, which receives the instance of the data-bound collection. When the binding expression is simply **`{Binding}`**, like in the previous example, the view takes the data source from the **`BindingContext`** property of its parent. This is going to be assigned in C#, and the reason for doing this in imperative code is that the most common approach is loading data in C# and then assigning data to the view. In the code-behind of the page, you could write the following code:

```
private ObservableCollection<Contact> Contacts { get; set; }
public ListViewSample()
{
 InitializeComponent();
 Contacts = new ObservableCollection<Contact>();
 Contact contact1 = new Contact
 {
  FirstName = “Alessandro",
  LastName = “Del Sole",
  DateOfBirth = new DateTime(1977, 05, 10),
  IsFamilyMember = false
 };
 Contact contact2 = new Contact
 {
  FirstName = “Robert",
  LastName = “White",
  DateOfBirth = new DateTime(1980, 02, 03),
  IsFamilyMember = true
 };
 Contact contact3 = new Contact
 {
  FirstName = “Angela",
  LastName = “Green",
  DateOfBirth = new DateTime(1982, 09, 01),
  IsFamilyMember = true
 };
 Contacts.Add(contact1);
 Contacts.Add(contact2);
 Contacts.Add(contact3);
 BindingContext = Contacts;
}
```

Assigning the collection instance to the **`BindingContext`** property will make such a collection the data source of the whole page. The **`ListView`**, via binding, will populate its **`ItemsSource`** property from the first container in the visual tree that has data in its **`BindingContext`** property. This means that the **`StackLayout`** is inheriting the **`BindingContext`** from the **`ContentPage`**, and the data passes through the visual tree.

**In the coming section about Model-View-ViewModel, you will learn a more efficient and logical approach to expose and bind data. For now, it is important to understand how binding collections works.**

The second consideration is an event called **`ItemSelected`**, which is fired when the user selects one of the items in the list. The event handler would look as follows:

```
 private void ContactList_ItemSelected(object sender,
 SelectedItemChangedEventArgs e)
 {
  var contact = e.SelectedItem as Contact;
  if(contact != null)
  {
   // Further code goes here
  }
 }
```

When the event is fired, an instance of the **`SelectedItemChangedEventArgs`** object is created and its **`SelectedItem`** property, of the object type, contains the selected item. This must be converted into the expected type. It is good practice to make the conversion via the as operator and then check whether the result is not null in order to avoid exceptions in case the conversion fails. The **`HasUnevenRows`** property set as true makes the height of rows dynamic.

The next key point is related to the fact that the **ListView**, as well as other views, does not know how you want to present the data. So, you need to implement a **DataTemplate**, which represents the set of visual elements you want to use to display data according to your needs. In the **ListView**, the **DataTemplate** always starts with a cell. Xamarin.Forms supports the cell types described in *Table 9.3*:

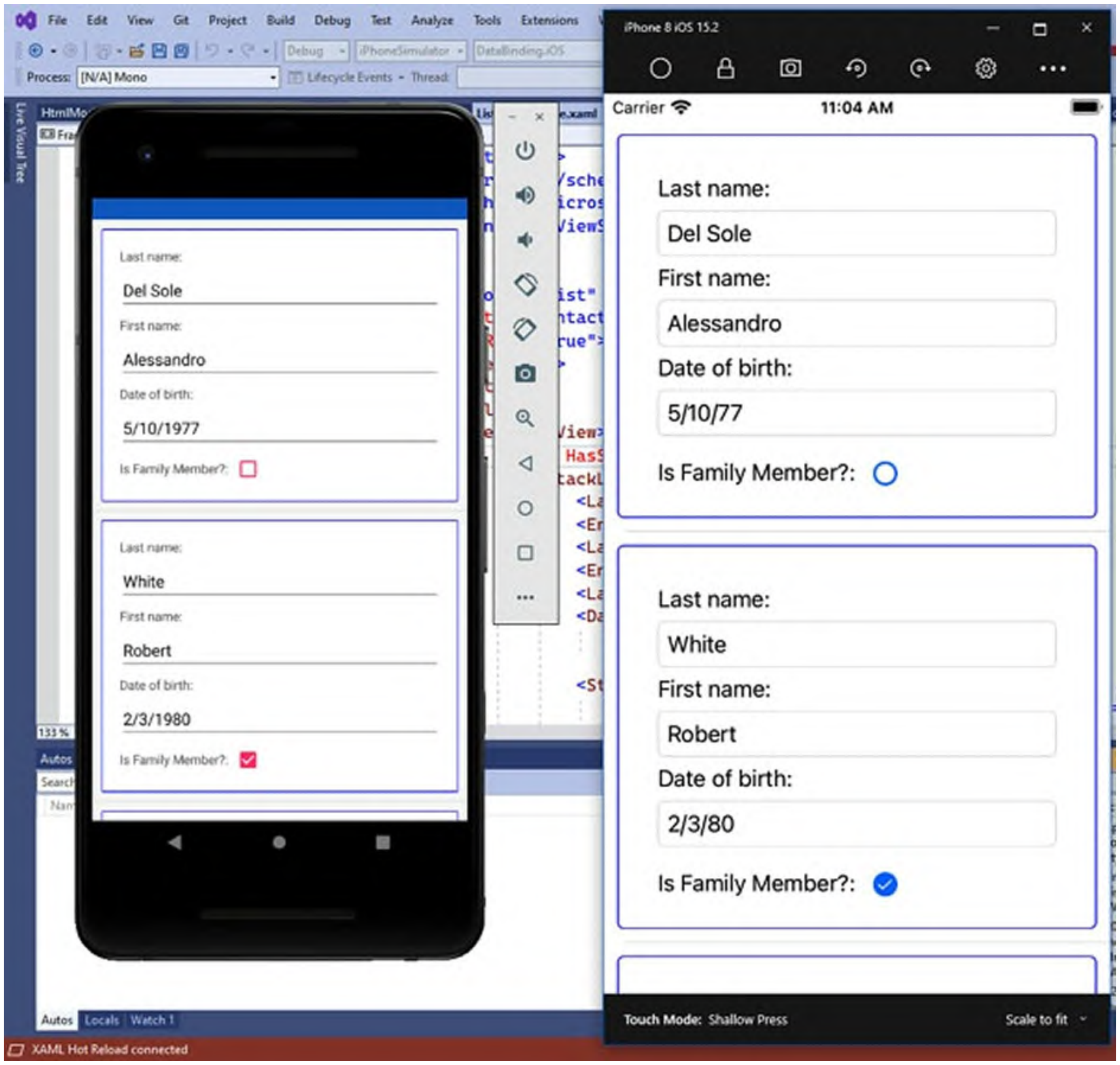
| Cell type | Description |
|---|---|
| **TextCell** | Displays two `Label` views: one with free text and one with a data-bound value. |
| **EntryCell** | Displays a `Label` with free text and an `Entry` with a data-bound string value. |
| **ImageCell** | Displays a `Label` with free text and an `Image` with a data-bound image. |
| **SwitchCell** | Displays a `Label` with free text and a `Switch` with a data-bound `bool` value. |
| **ViewCell** | Allows displaying data with custom visual elements, such as layouts. |

***Table 9.3:*** *Supported cell types*

The current example is based on a **ViewCell** because other cells would not be enough to present the information of the **Contact** class, so a custom layout is necessary. For example, if you only wanted to display (and edit) the last name, you could use an **EntryCell** type, as follows:

```
<StackLayout>
  <ListView x:Name="PeopleList" ItemsSource="{Binding}">
    <ListView.ItemTemplate>
      <DataTemplate>
        <EntryCell Label="Last name:" Text="{Binding LastName}"/>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</StackLayout>
```

Only one cell can be added to the **DataTemplate**, which means that you could not even combine multiple **EntryCell** types. This is another reason why the **ViewCell** is the most used cell. If you run the sample page, you will get the result shown in *Figure 9.3*:



Last name: Del Sole
First name: Alessandro
Date of birth: 5/10/1977
Is Family Member?: ☐

Last name: White
First name: Robert
Date of birth: 2/3/1980
Is Family Member?: ☑

Last name: Del Sole
First name: Alessandro
Date of birth: 5/10/77
Is Family Member?: ○

Last name: White
First name: Robert
Date of birth: 2/3/80
Is Family Member?: ●

***Figure 9.3:*** *Displaying and editing collections with the ListView*

As you can see, the **`ListView`** is presenting each instance of the **`Contact`** class in a separate row, and it allows for editing data via the views that you have specified in the data template. For example, if you tap on the date of birth field, you will see the **`DatePicker`** view in action. All the data that you enter will be automatically stored in the bound class.

**Tip: All the views that have built-in scrolling, like the `ListView` and the other views discussed in the next section, should never be enclosed inside a `ScrollView`; this is to avoid scrolling conflicts.**

## The DataTemplate as a resource

Views that allow for presenting lists expose the **`ItemTemplate`** property, which you assign with a **`DataTemplate`**, like in the previous example. However, data templates can be defined as reusable resources. For example, you could move the **`DataTemplate`** shown previously to the page's resources as follows:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <DataTemplate x:Key="ContactTemplate">
      <ViewCell>
        <ViewCell.View>
          <Frame HasShadow="False" BorderColor="Blue" Margin="10">
            <StackLayout>
            <Label Text="Last name:"/>
            <Entry Text="{Binding LastName}"/>
            <Label Text="First name:"/>
            <Entry Text="{Binding FirstName}"/>
            <Label Text="Date of birth:"/>
            <DatePicker Date="{Binding DateOfBirth,
                Mode=TwoWay}"/>
            <StackLayout Orientation="Horizontal">
              <Label Text="Is Family Member?:"
              VerticalOptions="Center"/>
              <CheckBox IsChecked="{Binding IsFamilyMember}"
              VerticalOptions="Center"/>
            </StackLayout>
            </StackLayout>
          </Frame>
        </ViewCell.View>
      </ViewCell>
    </DataTemplate>
  </ResourceDictionary>
</ContentPage.Resources>
```

You can then consume the data template by assigning in-line the **`ItemTemplate`** property of the view, as follows:

```
<ListView x:Name="ContactList" ItemsSource="{Binding}"
    ItemSelected="ContactList_ItemSelected"
    HasUnevenRows="True" ItemTemplate="{StaticResource
    ContactTemplate}" />
```

As you can see, you use the same syntax you saw with styles. There are several reasons to move data templates to reusable resources (following the same scope rules described in the *Understanding and defining resources* section), such as making them reusable across different views, or even different apps if written in a library, or making code more organized and easier to read.

## Binding different types: Value converters

Sometimes, you might need to bind data of a certain type but display them in another format. For better understanding, consider the current sample code (or follow the **`ConverterSamplePage.xaml`** file in the companion solution). Suppose you want to change the color of the Is Family Member? text in a **`Label`** based on the **`bool`** value of the **`IsFamilyMember`** property. You could bind the **`TextColor`** property of the **`Label`** to the **`IsFamilyMember`** property of the **`Contact`** class, but the first one is of type **`Color`** and the second one is of type **`bool`**. In these situations, you can use a value converter. This class implements the **`IValueConverter`** interface, which converts the input type into another type, returning the result to the user interface. Implementing a value converter is straightforward. Continuing the example, add a new code file to the shared project called **`BoolToColorConverter.cs`**. By convention, names of value converters, with the Converter suffix and their name, should make it easy to understand their purpose. The goal is to show the Is Family Member? text in red if the **`IsFamilyMember`** property is true or with the default system color if false. The following code demonstrates how to accomplish this (comments following):

```
using System;
using System.Globalization;
using Xamarin.Forms;
namespace DataBinding
{
  public class BoolToColorConverter : IValueConverter
  {
    public object Convert(object value, Type targetType,
      object parameter, CultureInfo culture)
    {
      try
      {
        bool originalValue = (bool)value;
        switch (originalValue)
        {
          case true:
            return Color.Red;
          default:
            return Color.Default;
        }
      }
      catch (Exception)
      {
       return Color.Default;
      }
    }
    public object ConvertBack(object value, Type targetType,
      object parameter, CultureInfo culture)
    {
      throw new NotImplementedException();
    }
  }
}
```

The **`IValueConverter`** interface requires a converter class to implement two methods: **`Convert`** and **`ConvertBack`**. **`Convert`** allows you to convert the source type into another type that the view expects, whereas **`ConvertBack`** does exactly the opposite. In this case, the code first converts the value parameter from object to **`bool`** and then, depending on its value, returns a different color. You do not need to implement **`ConvertBack`** if your binding direction is read-only, like in this case, where your views do not edit the converted value, they just use it. Both methods receive the following four parameters, even if only the first and third ones are supplied by the developer:

- **`value`**: The source object that needs to be converted.
- **`targetType`**: The expected type of the source, as detected by the compiler.
- **`parameter`**: An additional parameter that you can pass from the binding expression to supply data to the converter.
- **`culture`**: This is also inferred by the compiler and is represented by an instance of the **`CultureInfo`** class containing the localization information for the current system.

The next step is to consume the converter in XAML. You first need to add an XML namespace declaration in order to reference the C# namespace that exposes the converter class, like in the following example:

```
xmlns:local="clr-namespace:DataBinding"
```

The next step is adding a resource that references the converter, as follows:

```
<ContentPage.Resources>
  <ResourceDictionary>
```

```
    <local:BoolToColorConverter x:Key="BoolToColorConverter" />
    ……
  </ResourceDictionary>
 <ContentPage.Resources>
```

Now, the converter is assigned to the **Converter** property of the binding expressions of interest, as shown in the following code:

```
 <StackLayout Orientation="Horizontal">
  <Label Text="Is Family Member?:"
    TextColor="{Binding IsFamilyMember, Converter=
    {StaticResource
        BoolToColorConverter}}"
    VerticalOptions="Center"/>
  <CheckBox IsChecked="{Binding IsFamilyMember}"
    VerticalOptions="Center"
    Color="{Binding IsFamilyMember, Converter={StaticResource
      BoolToColorConverter}}"/>
 </StackLayout>
```

As the converter is a resource, the reference is added using the **StaticResource** markup extension. In both views, properties of the **Color** type are bound to the **IsFamilyMember** property, which is of the bool type, but the converter returns an object of the **Color** type based on the **bool** value. If you run the code, you will see the result shown in *Figure 9.4*:

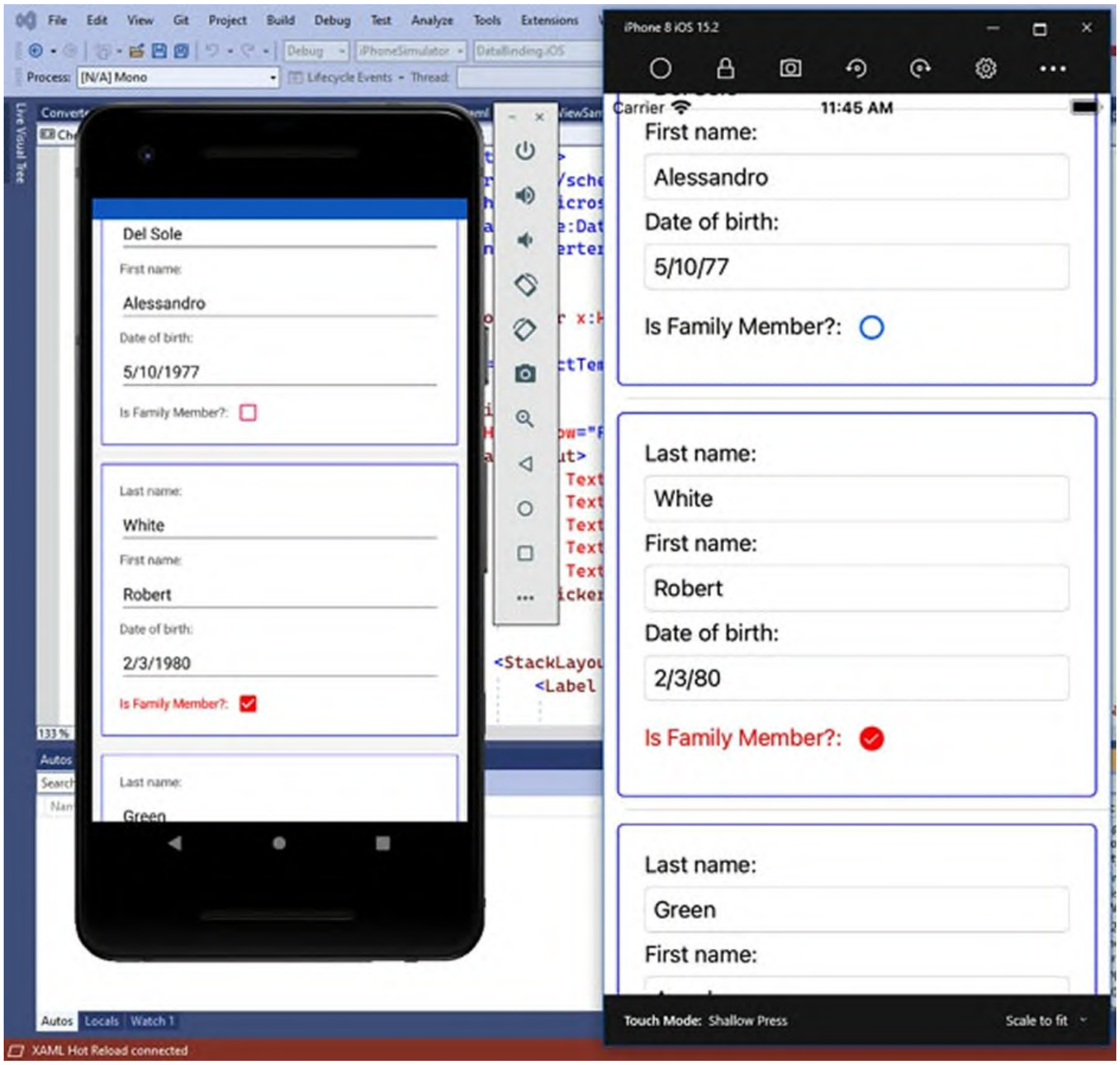


***Figure 9.4:** Consuming value converters*

As you can see, the text and the checkbox are shown in red if the **`IsFamilyMember`** property of the bound **`Contact`** class is true. Additionally, you can remove the flag and see how everything returns to the default color, which also demonstrates that data binding is also working. If you need to pass a parameter to the converter for further logic implementation, the syntax you use is as follows:

```
{Binding PropertyName, Converter={StaticResource ConverterName},
ConverterParameter=value}}
```

The following is a list of example scenarios where you might need to implement value converters:

- Formatting **`DateTime`** objects into custom string representations.
- Displaying HTML string values into a **`WebView`**.
- Converting a byte array into an **`ImageSource`** to display an image.
- Converting an integer value to a **`Boolean`** true/false representation.

Value converters are a powerful feature, and they are widely used in real-world development. You will use them very often, and other examples will be provided in the upcoming sections.

## Displaying Collections efficiently

With the most recent versions of Xamarin.Forms, several views have been added to display collections with different user experiences. This section describes such additional views, and it also discusses how to show lists within the built-in system picker.

### Displaying lists with the CollectionView

In the past, displaying and binding lists of data in Xamarin.Forms was only possible with the **`ListView`**. Starting with version 4.0, Xamarin.Forms provides a new view to display and bind lists: **`CollectionView`**. This view has the following advantages over the **`ListView`**:

- It has a simpler API surface.
- It does not need cells to implement data templates, rendering the user interface a lighter process than before.
- It exposes properties that allow the customizing of some layout points without the need to write native code, unlike the **`ListView`**.

**The `CollectionView` is a powerful and modern view that should be considered as an alternative to the `ListView`, rather a replacement.**

If you consider the same collection of **`Contact`** class, you could rewrite the user interface using the **`CollectionView`** as follows:

```
<?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="DataBinding.CollectionViewSample"
    Padding="0,20,0,0">
```

```
  <ContentPage.Content>
   <StackLayout>
     <CollectionView x:Name="ContactList"
         ItemsSource="{Binding Contacts}" Margin="15"
         SelectionMode="Single"
         SelectionChanged="ContactList_ SelectionChanged"
         VerticalScrollBarVisibility="Never"
         HorizontalScrollBarVisibility="Never">
      <CollectionView.ItemTemplate>
       <DataTemplate>
       <StackLayout Margin="10">
        <Label Text="Full name:"/>
        <Entry Text="{Binding LastName}"/>
        <Label Text="First name:"/>
        <Entry Text="{Binding FirstName}"/>
        <Label Text="Date of birth:"/>
        <DatePicker Date="{Binding DateOfBirth,
          Mode=TwoWay}"/>
        <StackLayout Orientation="Horizontal">
         <Label Text="Is Family Member?:"
          VerticalOptions="Center"/>
         <CheckBox IsChecked="{Binding IsFamilyMember}"
          VerticalOptions="Center"/>
        </StackLayout>
       </StackLayout>
       </DataTemplate>
      </CollectionView.ItemTemplate>
     <CollectionView.EmptyView>
      <Label Text="No data is available" TextColor="Red"
        FontSize="Large"
        VerticalOptions="Center" VerticalTextAlignment="Center"
        HorizontalOptions="Center"
        HorizontalTextAlignment="Center"/>
     </CollectionView.EmptyView>
    </CollectionView>
   </StackLayout>
  </ContentPage.Content>
 </ContentPage>
```

In the definition of the **`CollectionView`**, you can see how the bound collection is still assigned via the **`ItemsSource`** property. In addition, you find the following elements:

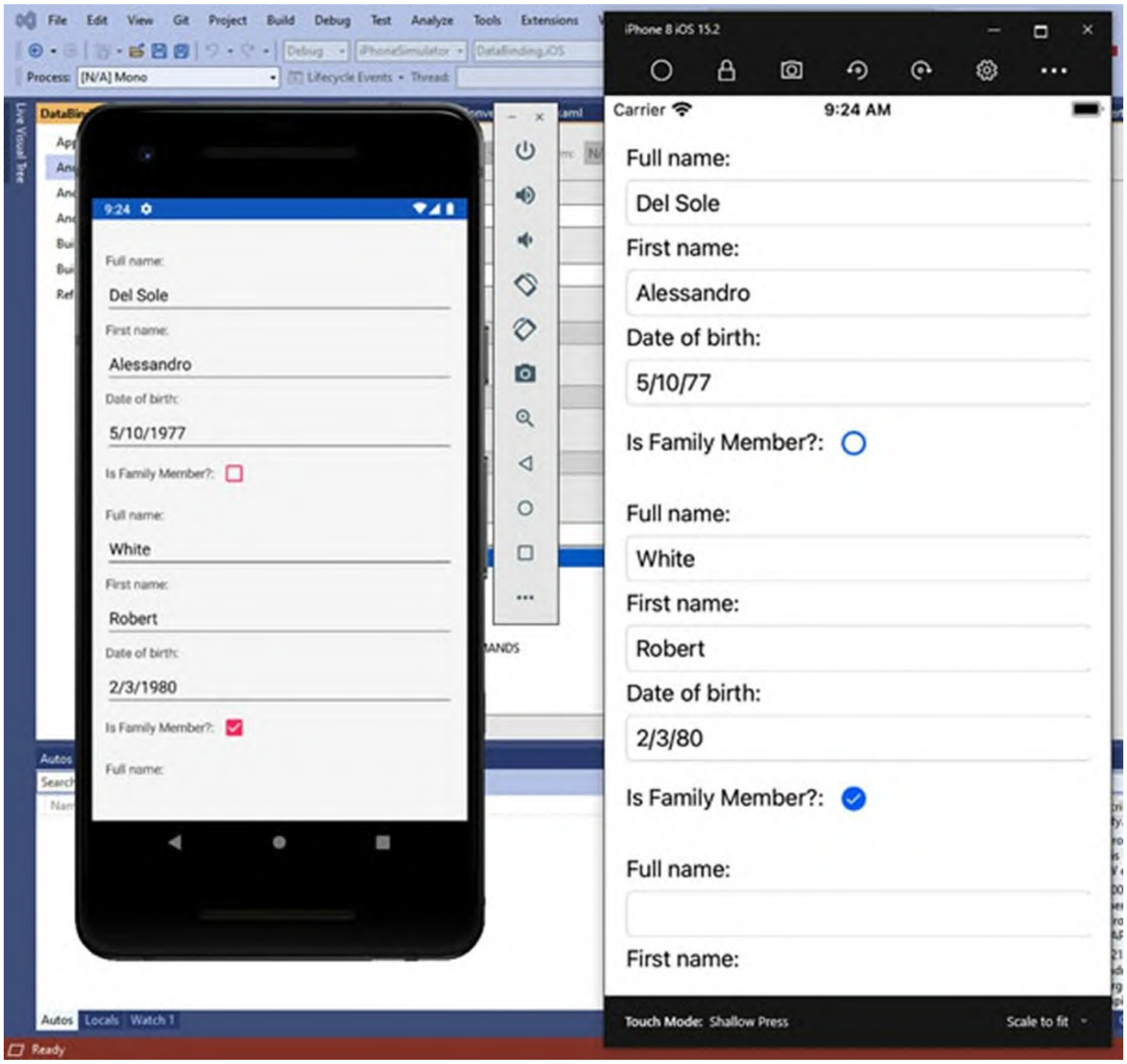
- **`SelectionMode`**: A property that specifies if the user can select one (single) or more (multiple) items. If set to None, the selection is disabled. In the **`ListView`**, you need to write a custom renderer to disable the selection.

- **`HorizontalScrollBarVisibility`** and **`VerticalScrollBarVisibility`**: These properties allow for setting the visibility of the horizontal and vertical scrollbars. Supported values are **`Always`**, **`Never`**, and **`Default`**. The latter takes the target OS default behavior.
- **`SelectionChanged`**: An event that is raised when the user selects one or more items in the list, depending on the value of **`SelectionMode`**.

The way items are presented is provided via the **`DataTemplate`**, like for the **`ListView`**, but in this case, you no longer need cells, and you can go for a layout. At the bottom of the previous XAML you can also see how the **`CollectionView`** exposes another useful property: **`EmptyView`**. This can be assigned with an individual view or layout that is automatically shown when the bound collection is null or empty. This is extremely useful because it eliminates the need for you to write a custom user interface that is displayed when the collection is empty, which is something you need to do when working with the **`ListView`**. Before running the example, you need to data-bind a collection to the user interface in the code-behind file for the page, as follows:

```
private ContactViewModel ViewModel { get; set; }
public CollectionViewSample()
{
  InitializeComponent();
  ViewModel = new ContactViewModel();
  BindingContext = ViewModel;
}
```

*Figure 9.5* shows the result of the current code:



9:24

Full name:
Del Sole
First name:
Alessandro
Date of birth:
5/10/1977
Is Family Member?: ☐
Full name:
White
First name:
Robert
Date of birth:
2/3/1980
Is Family Member?: ☑
Full name:

Carrier 9:24 AM

Full name:
Del Sole
First name:
Alessandro
Date of birth:
5/10/77
Is Family Member?: ○
Full name:
White
First name:
Robert
Date of birth:
2/3/80
Is Family Member?: ✓
Full name:

First name:

*Figure 9.5: Displaying data with the CollectionView*

In terms of appearance, there are additional benefits to using the **`CollectionView`**. By default, items are presented vertically, and each row contains one item, but you can quickly change this behavior. For example, you can easily define a horizontal list view as follows:

```
<CollectionView.ItemsLayout>
  <LinearItemsLayout Orientation="Horizontal"/>
</CollectionView.ItemsLayout>
```

The **`ItemsLayout`** property can be set with two objects: **`LinearItemsLayout`** and **`GridItemsLayout`**. With **`LinearItemsLayout`**, you can make the **`CollectionView`** display one item per row, either vertically or horizontally. When **`LinearItemsLayout`** is not specified, vertical orientation is assumed, like in the previous code listing. The **GridItemsLayout** object makes it possible to create a grid view, as follows:

```
<CollectionView.ItemsLayout>
  <GridItemsLayout Orientation="Vertical" Span="3"/>
</CollectionView.ItemsLayout>
```

You just need to specify the **Orientation** (**Horizontal** or **Vertical**) and the number of items per row, via the **Span** property. Obviously, you need to consider the size of each item you need to present and the available space on screen. *Figure 9.6* shows an example of vertical grid view:

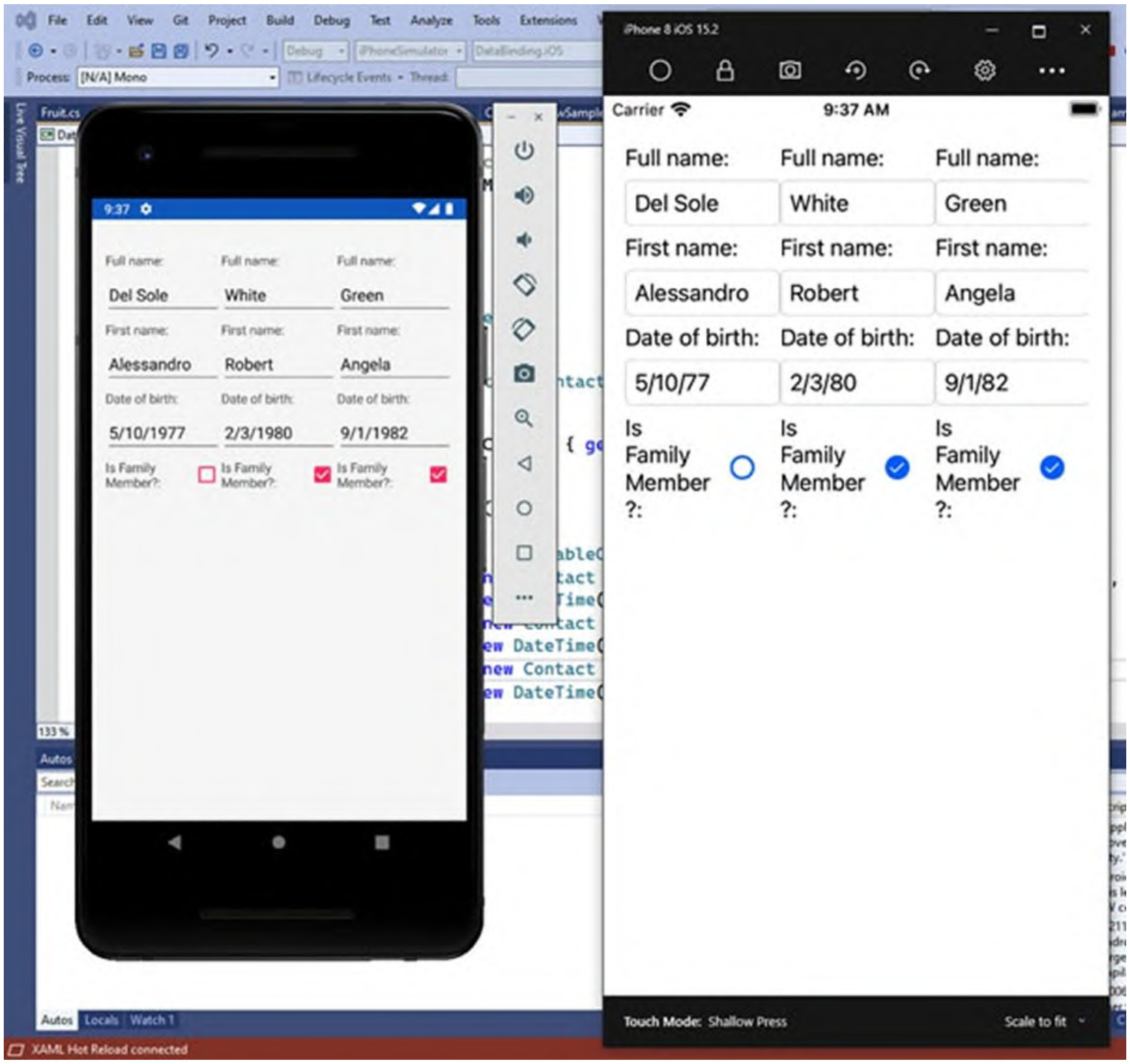


***Figure 9.6:*** *Implementing a grid view*

It is also necessary to comment on how you handle the **SelectionChanged** event. The following code demonstrates this:

```
private void ContactList_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
  // In case of single selection
  var selectedPerson = this.ContactList.SelectedItem as Contact;
  // In case of multi-selection:
  var singlePerson = e.CurrentSelection.FirstOrDefault() as
  Contact;
  var selectedObjects = e.CurrentSelection.Cast<Contact>();
  foreach (var person in selectedObjects)
  {
   // Handle your object properties here...
  }
}
```

If **`SelectionMode`** is assigned with **`Single`**; you just need the first line of code in the event handler, which is something you already saw for the **`ListView`**. If **`SelectionMode`** is assigned with **`Multiple`**, you need to invoke a generic method called **`Cast`** over the **`CurrentSelection`** property (of the **`IReadOnlyList<object>`** type) of the **`SelectionChangedEventArgs`** object instance, passing the target type as the **`type`** parameter. **`Cast`** converts each object in the collection into a collection of the appropriate type. Being a modern and simple-to-use data view, the **`CollectionView`** has quickly entered the code base of thousands of projects, and it is also recommended that you use this with new projects.

## Scrolling lists with the CarouselView

The **`CarouselView`** is another view that allows for scrolling lists, with additional members that specifically support swipe gestures. Behind the scenes, the **`CarouselView`** is built on top of the **`CollectionView`**, and it allows for scrolling lists both horizontally and vertically. For a better understanding, first consider *Figure 9.7*, where you can see items being scrolled inside a **`CarouselView`** by swiping the view:

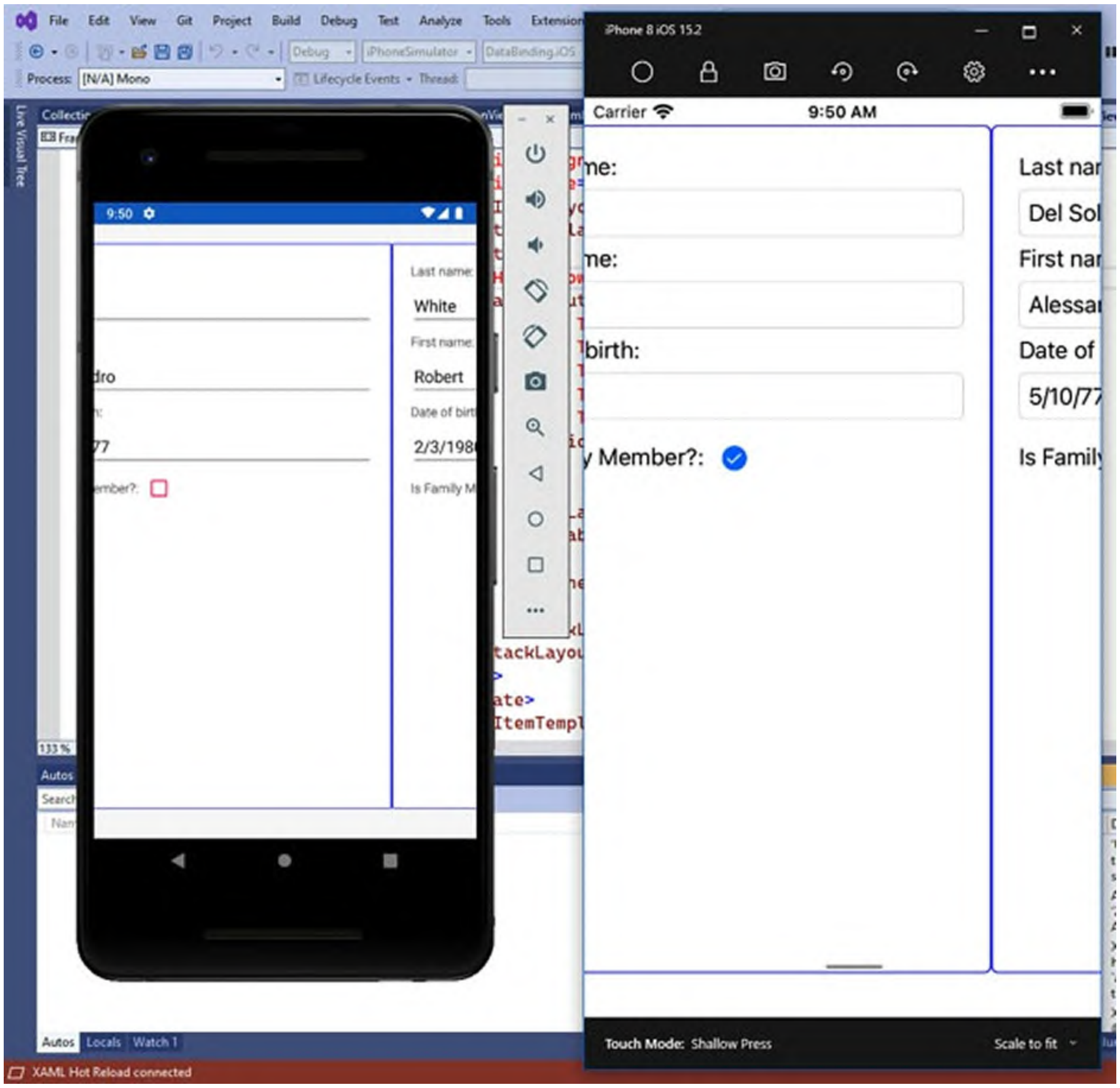


***Figure 9.7:*** *Swiping items in a CarouselView*

The following code demonstrates how to implement a **`CarouselView`**:

```
<?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="DataBinding.CarouselViewSample"
    Padding="0,20,0,20">
  <ContentPage.Content>
    <StackLayout Orientation="Vertical">
      <CarouselView x:Name="PeopleList" ItemsSource="{Binding
      Contacts}"
          CurrentItemChanged="PeopleList_CurrentItemChanged"
          PositionChanged="PeopleList_PositionChanged"
          CurrentItem="{Binding SelectedContact}">
      <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal"
          SnapPointsAlignment="Center"
          SnapPointsType="Mandatory"/>
      </CarouselView.ItemsLayout>
      <CarouselView.ItemTemplate>
        <DataTemplate>
          <Frame HasShadow="False" BorderColor="Blue" Margin="20">
          <StackLayout>
            <Label Text="Last name:"/>
            <Entry Text="{Binding LastName}"/>
            <Label Text="First name:"/>
            <Entry Text="{Binding FirstName}"/>
            <Label Text="Date of birth:"/>
            <DatePicker Date="{Binding DateOfBirth,
              Mode=TwoWay}"/>
            <StackLayout Orientation="Horizontal">
              <Label Text="Is Family Member?:"
              VerticalOptions="Center"/>
              <CheckBox IsChecked="{Binding IsFamilyMember}"
                VerticalOptions="Center"/>
            </StackLayout>
          </StackLayout>
          </Frame>
        </DataTemplate>
      </CarouselView.ItemTemplate>
     </CarouselView>
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

There are a few differences when compared to the **`CollectionView`**. First of all, the **`PositionChanged`** event is raised when the view is scrolled. The position is returned as an **`int`**, exposed by the **`CurrentPosition`** property of the **`PositionChangedEventArgs`** object:

```
private void PeopleList_PositionChanged(object sender,
  PositionChangedEventArgs e)
{
  int currentPosition = e.CurrentPosition;
  int previousPosition = e.PreviousPosition;
}
```

It is also easy to retrieve the previous position via the **`PreviousPosition`** property of the **`int`** type. Instead, the **`CurrentItemChanged`** event is raised when the user selects an item in the view. You can retrieve the related information as follows:

```
private void PeopleList_CurrentItemChanged(object sender,
  CurrentItemChangedEventArgs e)
 {
  var currentItem = e.CurrentItem as Contact;
  var previousItem = e.PreviousItem as Contact;
 }
```

Both the **`CurrentItem`** and **`PreviousItem`** properties are of type object, so you need to convert them to the expected type. You have further control over the view's appearance via the **`ItemsLayout`** property and its **`LinearItemsLayout`** member, whose value can be **`Horizontal`** (default) or **`Vertical`**. You can also control snap points via the **`SnapPointType`** property. These represent how the scrolled items should behave, for example, by making them scroll naturally with inertia (**`Mandatory`**) or by making them scroll one at a time (**`MandatorySingle`**). Snap points can be disabled by assigning **`None`**. Instead, the **`SnapPointsAlignment`** property specifies how snap points are aligned to items, and it can be assigned with **`Start`**, **`Center`**, or **`End`**. The **`CarouselView`** allows for implementing a very common user experience, but most of the times, it is also recommended to display indicators that show how many items are in the list and what is the current item. In the past, you had to do all this manually, but with the latest versions of Xamarin.Forms, you can take advantage of the **`IndicatorView`**.

## Displaying item indicators with the IndicatorView

The **`IndicatorView`** allows for displaying the number of items and the current item in a **`CarouselView`**. For better understanding, look at the indicator at the bottom of the page in *Figure 9.8*:

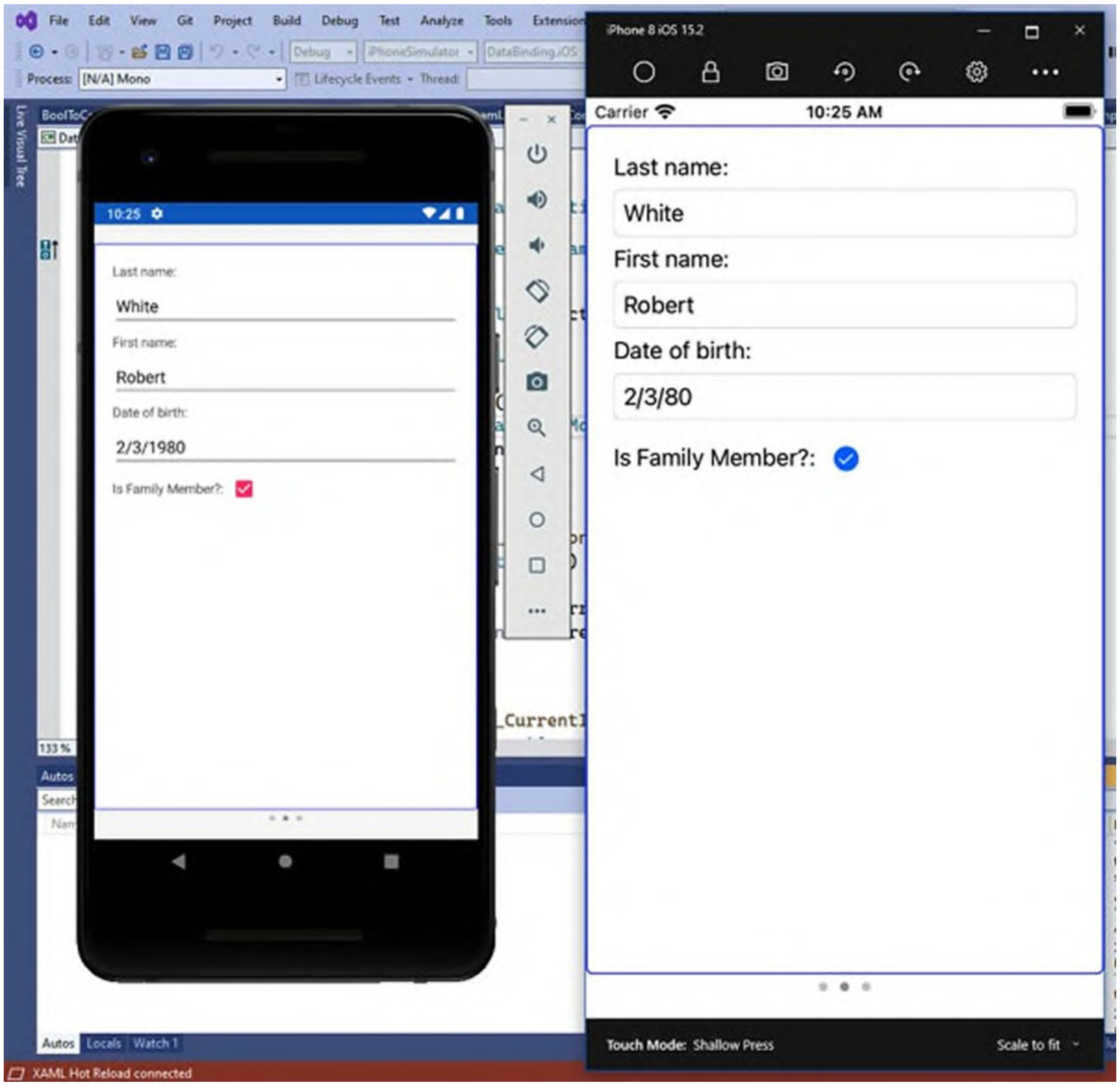


***Figure 9.8:*** *Displaying indicators with the IndicatorView*

Implementing the **`IndicatorView`** is very simple. You can add one in your visual tree with a simple declaration like the following:

```
 <IndicatorView x:Name="PersonIndicatorView"
   IndicatorColor="LightGray"
   SelectedIndicatorColor="DarkGray"
   HorizontalOptions="Center" />
```

Note how you specify colors of the general indicator with the **`IndicatorColor`** property and the color of the current indicator with the **`SelectedIndicatorColor`** property. It is important to mention that a name must be assigned to the view because you will add a reference to the **`IndicatorView`** by passing its name to the **`IndicatorView`** property of the

**CarouselView**, as follows:

```
<CarouselView x:Name="PeopleList" ItemsSource="{Binding
Contacts}"
    CurrentItemChanged="PeopleList_ CurrentItemChanged"
    PositionChanged="PeopleList_PositionChanged"
      IndicatorView="PersonIndicatorView"
    CurrentItem="{Binding SelectedContact}">
```

There is nothing else you must do. The **CarouselView** and the **IndicatorView** will stay connected, and the latter will be able to display the current indicator based on the current item of the **CarouselView**.

## Selecting items with the Picker

The **Picker** is a Xamarin.Forms view that allows for selecting an item from a list and that maps the target system's selector control. For better understanding, consider *Figure 9.9*, which demonstrates how to select one fruit from a list with the **Picker**:

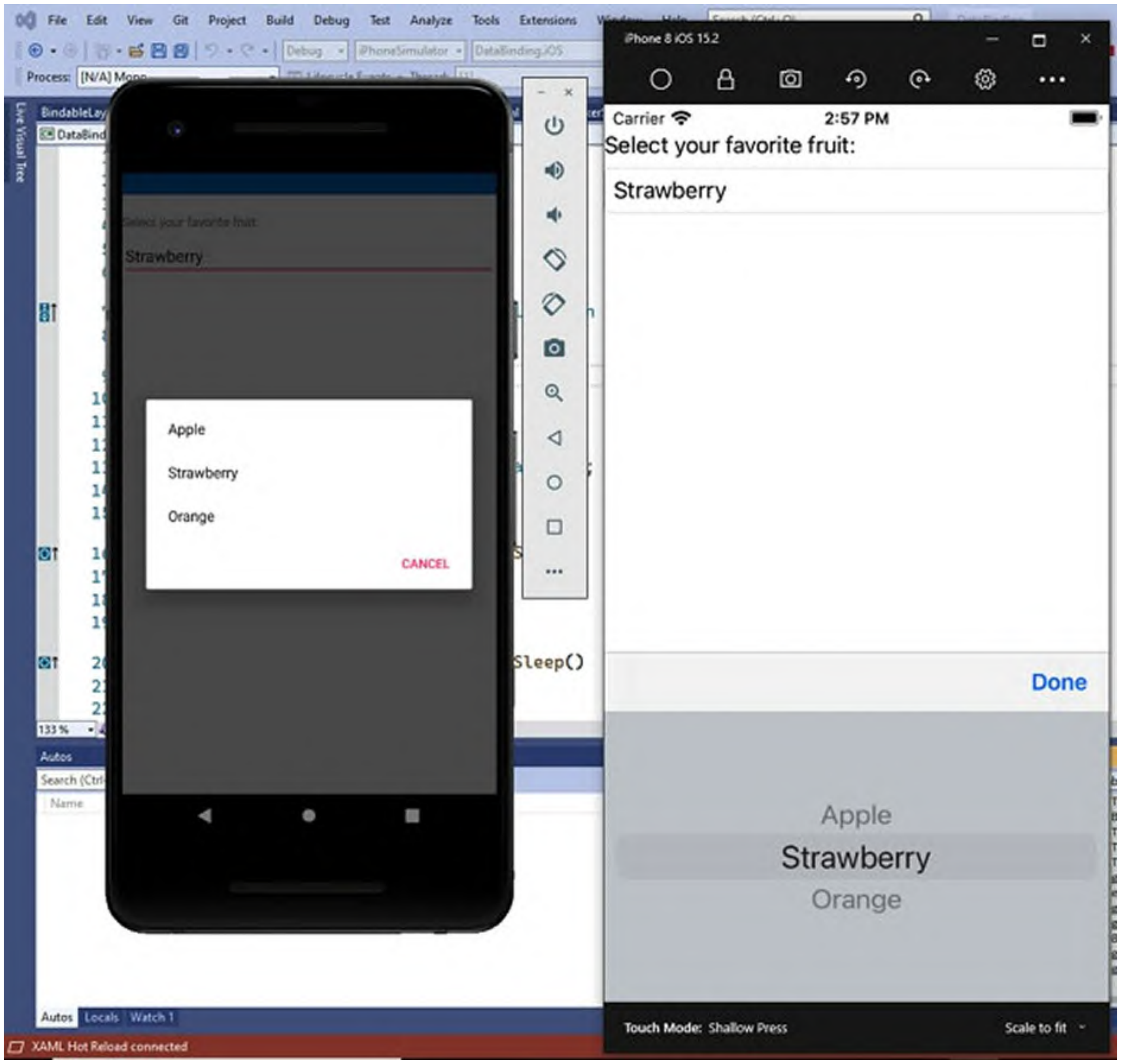


***Figure 9.9:*** *Item selection with the Picker*

As you can see, the **`Picker`** invokes the system's selector. It exposes an **`ItemsSource`** property, like the previously discussed views, that is assigned with a collection that implements the **`IEnumerable`** interface. The following XAML code shows how to implement the sample **`Picker`**:

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

The appropriate comments will help you understand how the bound collection is implemented. In the constructor of the page, you can have the

following code:
```
public PickerSample()
{
  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;
}
```
As you can see, the code defines some objects of the **`Fruit`** type, a class declared as follows:
```
public class Fruit
{
  public string Name { get; set; }
  public string Color { get; set; }
}
```
Next, a new **`ObservableCollection<Fruit>`** is declared and populated with all the objects. Finally, the **`ItemsSource`** property of the **`Picker`** is assigned with such a collection. Each **`Fruit`** object has two properties, **`Name`** and **`Color`**, but the **`Picker`** does not know which to display in the selection view. For this reason, the **`ItemDisplayBinding`** property of the **`Picker`** is assigned with the name of the property of the bound object that will be used to display values in the list (see the preceding XAML code). When the user selects an item in the list, the **`Picker`** fires an event called **`SelectedIndexChanged`**. The name of the event makes it clear that this view works with the item position rather than with the item instance. However, you do not have to deal with indexes and item positions. In fact, you can quickly get the selected item, as follows:
```
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");
}
```
In short, you convert the value of the **`Picker.SelectedItem`** property into an object of the expected type, **`Fruit`** in this case. It is good practice to make the

conversion via the as operator so that a failing conversion will return null instead of throwing an exception, and you will be able to check whether the conversion result is not null with a simple if block. Once you have the converted object, you can manipulate it according to your needs.

## Introducing bindable layouts

Xamarin.Forms allows any layout to support data binding via the so-called **bindable layouts** feature. Suppose you want to bind to a **`StackLayout`** the same collection of **`Contact`** objects used in the previous examples about the **`ListView`** and **`CollectionView`**. You can write the following XAML:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBinding.BindableLayoutsSamplePage">
  <ContentPage.Content>
   <StackLayout BindableLayout.ItemsSource="{Binding Contacts}">
    <BindableLayout.ItemTemplate>
     <DataTemplate>
      <StackLayout Margin="10">
       <Label Text="Full name:"/>
       <Entry Text="{Binding LastName}"/>
       <Label Text="First name:"/>
       <Entry Text="{Binding FirstName}"/>
       <Label Text="Date of birth:"/>
       <DatePicker Date="{Binding DateOfBirth,
           Mode=TwoWay}"/>
       <StackLayout Orientation="Horizontal">
       <Label Text="Is Family Member?:"
        VerticalOptions="Center"/>
       <CheckBox IsChecked="{Binding IsFamilyMember}"
           VerticalOptions="Center"/>
       </StackLayout>
      </StackLayout>
     </DataTemplate>
    </BindableLayout.ItemTemplate>
    <BindableLayout.EmptyView>
     <Label Text="No data is available" TextColor="Red"
     FontSize="Large"
         VerticalOptions="Center" VerticalTextAlignment="Center"
         HorizontalOptions="Center"
         HorizontalTextAlignment="Center"/>
    </BindableLayout.EmptyView>
   </StackLayout>
  </ContentPage.Content>
```

```
</ContentPage>
```

Here's a summary of the most relevant points:

- Layouts can leverage properties from the `BindableLayout` object, which enables collection data binding on layouts.
- The `BindingLayout.ItemsSource` property is assigned to a data collection, exactly as you would do with a `ListView` or `CollectionView`.
- You use the `BindingLayout.ItemTemplate` property to define a data template to display items in the collection, again, exactly as you would do with `ListView` or `CollectionView`.
- With the `BindableLayout.EmptyView`, you can specify a view that will be displayed if the bound collection is empty.

Data binding then works exactly like in other views described previously.

**Bindable layouts have very poor rendering performance. For this reason, you should avoid them when possible and prefer the `CollectionView` or `ListView` views, even with very short lists. Based on the experience about fixing performance issues in several projects, a `CollectionView` can render in less than 1 second, what a bindable layout can render in up to 7 or 8 seconds.**

## Advanced data binding: The Model-View-ViewModel pattern

The Model-View-ViewModel pattern, or MVVM, is an architectural pattern used with XAML-based development platforms, like Xamarin.Forms, WPF, and UWP. It provides strong and clear separation between the layers of an application. For the sake of simplicity, you will transform the example provided previously about a collection of `Contact` objects displayed inside a `CollectionView` into code based on MVVM. The model is the data. The viewmodel represents the business logic. The view is the user interface. Based on these assumptions, the `Contact` class is the model. The business logic will be encapsulated inside a new viewmodel class, which will expose a collection of `Contact` objects and actions over the data. The view part will be defined inside an XAML page. With MVVM, XAML pages and their code-behind files can only contain code that is related to the user interface and data binding assignments, nothing else. There are several benefits of using MVVM, especially in large projects. The following are the most important ones:

- Changes can be made to the user interface, in XAML, without affecting any code that works against data.
- The business logic in the viewmodel should be able to work with objects without knowing where they come from. This makes it possible to change the data access layer (for example, from an SQLite database to a remote data service) without changing the logic.

MVVM can be quite complex, but the more you work with it, the more you will appreciate its benefits. There are several libraries in the market that simplify the way you implement MVVM, and they will be listed at the end of this section. That said, you need to know the principles behind MVVM, so you will not use them here.

**Tip: It is good practice to keep the code well organized by placing models into a subfolder called *Models*, viewmodels into a subfolder called `ViewModels`, and views into a subfolder called *Views*. You can also create additional subfolders to better organize models, viewmodels, and views by area.**

## Defining the data model

In MVVM, the model represents the data. In C#, the data is represented by a class. For example, the **`Contact`** class used earlier is the model for the current example. Its code is relisted here for your reference:

```
public class Contact : INotifyPropertyChanged
{
 private string _lastName;
 public string LastName
 {
  get
  {
   return _lastName;
  }
  set
  {
```

```
  _lastName = value;
  OnPropertyChanged();
 }
}
private string _firstName;
public string FirstName
{
 get
 {
  return _firstName;
 }
 set
 {
  _firstName = value;
  OnPropertyChanged();
 }
}
private DateTime dateOfBirth;
public DateTime DateOfBirth
{
 get
 {
  return dateOfBirth;
 }
 set
 {
  dateOfBirth = value;
  OnPropertyChanged();
 }
}
private bool _isFamilyMember;
public bool IsFamilyMember
{
 get
 {
  return _isFamilyMember;
 }
 set
 {
  _isFamilyMember = value;
  OnPropertyChanged();
 }
}
public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged([CallerMemberName]
  string propertyName = null)
{
 PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }
  }
```

What you need to remember is that your models should implement the **INotifyPropertyChanged** interface in order to support change notifications. By default, when a new instance of the class is created, the **DateOfBirth** property is assigned the 1st of January 1900. You might want to change the default to a different date by adding a constructor that looks like this:

```
public Contact()
{
  DateOfBirth = DateTime.Today;
  IsFamilyMember = true;
}
```

The default date is assigned with today's date, and the **IsFamilyMember** is also assigned with true for every new instance. The reason is that you will implement queries that only return family members, as described in the upcoming section.

## Implementing the business logic: Commands and ViewModels

Viewmodels are a crucial part of the pattern. A viewmodel class only knows the data type of the model, but it does not know where the data comes from and, most importantly, it does not know which view will use the viewmodel itself. This means that a viewmodel must work in the most abstract way possible. A viewmodel has two main purposes: exposing data objects to views and implementing actions that will manipulate data based on your business logic. The reason why views pass through viewmodels to access data, instead of accessing data directly, is to filter actions and make sure data is processed by an intermediate layer that knows your business logic. The first thing you need to do is create a viewmodel class whose name ends with **ViewModel** by convention, such as **ContactViewModel**. This class must also implement the **INotifyPropertyChanged** interface.

### Exposing data

The first part of the sample viewmodel class exposes a collection, a single instance of the current **Contact** class, and implements change notification:

```
public class ContactViewModel: INotifyPropertyChanged
```

```
{
 public ObservableCollection<Contact> Contacts { get; set; }
 private Contact _selectedContact;
 public Contact SelectedContact
 {
  get
  {
   return _selectedContact;
  }
  set
  {
   _selectedContact = value;
   OnPropertyChanged();
  }
 }
 public event PropertyChangedEventHandler PropertyChanged;
 private void OnPropertyChanged([CallerMemberName] string
 propertyName = null)
 {
  PropertyChanged?.Invoke(this,
   new PropertyChangedEventArgs(propertyName));
 }
```

The next step is loading data. In the current example, data is created in code. In the last section of this chapter, you will learn how to load data from an SQLite database. The following method demonstrates this:

```
  private void LoadSampleData()
  {
   Contacts = new ObservableCollection<Contact>();
   // sample data
   Contact person1 =
    new Contact
    {
        FirstName = “Alessandro",
        LastName="Del Sole",
        IsFamilyMember = false,
     DateOfBirth = new DateTime(1977, 5, 10)
    };
   Contact person2 =
    new Contact
    {
     FirstName = “Robert",
     LastName="White",
     IsFamilyMember = false,
     DateOfBirth = new DateTime(1960, 2, 1)
    };
   Contact person3 =
    new Contact
    {
     FirstName = “Joseph",
     LastName = “Green",
     IsFamilyMember = true,
     DateOfBirth = new DateTime(1980, 4, 2)
    };
   Contacts.Add(person1);
   Contacts.Add(person2);
   Contacts.Add(person3);
  }
```

There is nothing difficult in this code, which creates new contacts and populates the collection.

## Defining actions with Commands

The next step is defining the actions that any caller view will be able to invoke. For example, adding or deleting items from the collection can be considered actions. To accomplish this, you define commands. A command is an object of the **`Command`** type, and views can access it via data binding, as you will see in the upcoming sections. Commands must be defined as properties, like in the following example that defines two commands: one for adding and one for deleting contacts in the collection:

```
 public Command AddCommand { get; set; }
 public Command DeleteCommand { get; set; }
```

As a second step, command properties must be assigned with an instance of the **`Command`** class, whose constructor needs the action that must be executed when the command is invoked, and optionally, a bool condition that specifies whether the action can be executed. The following code for the constructor of the viewmodel demonstrates this. Note that the code also invokes the **`LoadSampleData`** method to load data and includes the closing bracket for the class:

```
   public ContactViewModel()
   {
    LoadSampleData();
    AddCommand =
      new Command(() => Contacts.Add(new Contact()));
    DeleteCommand =
      new Command<Contact>((contact) => Contacts.Remove(contact),
      (contact) => contact != null);
   }
 }
```

An instance of the **Command** class takes at least one argument, **Execute**, of the **Action<T>** type representing the action that is executed when the command is invoked. Optionally, you can specify the **CanExecute** parameter, of type bool, which determines whether the command can be executed. Consider the **AddCommand** object first. It is assigned with an instance of the **Command** class, whose first parameter is an **Action<T>**, representing the action that is executed against the data. More specifically, the **Action<T>** is syntactically represented by a lambda expression, whose body adds a new **Contact** instance to the collection. For the **DeleteCommand** object, there is one more step. In fact, a bool condition is specified to address the **CanExecute** parameter. In this case, the action will be executed only if an instance of the **Contact** class is passed to the command. Now, you have a fully functional viewmodel that needs to be invoked by some views.

## Designing the user interface

The sample user interface will be made of two major components: a **CollectionView** that displays the list of contacts, and a toolbar with two buttons that invoke the commands defined in the viewmodel. The following XAML demonstrates this:

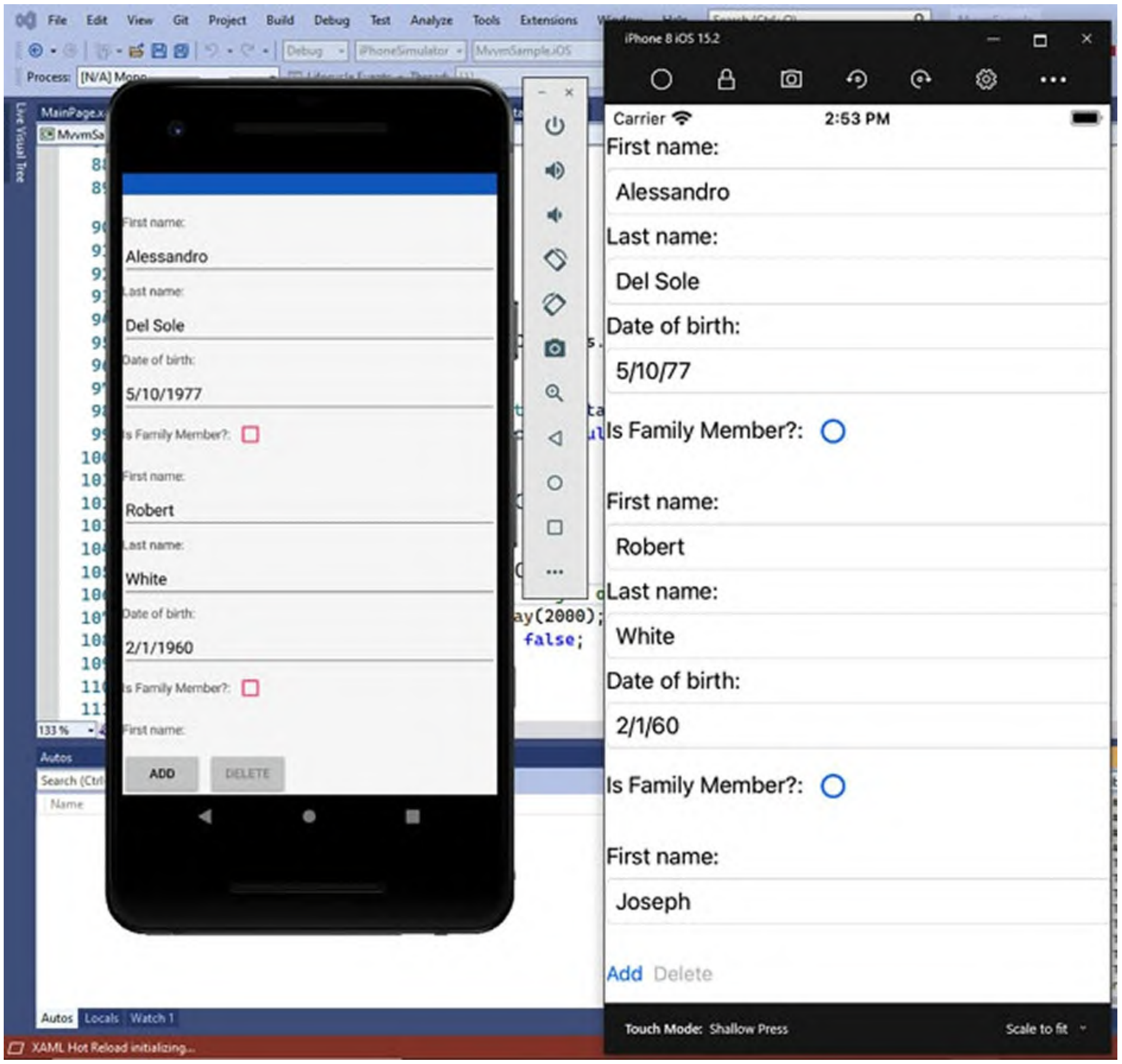
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MvvmSample.MainPage" Padding="0,20,0,0">
  <StackLayout>
   <CollectionView x:Name="ContactsList"
     ItemsSource="{Binding Contacts}"
     SelectionMode="Single"
     SelectedItem="{Binding SelectedContact}"
     VerticalScrollBarVisibility="Never"
     HorizontalScrollBarVisibility="Never">
    <CollectionView.ItemTemplate>
     <DataTemplate>
      <StackLayout Margin="10">
       <Label Text="First name:"/>
       <Entry Text="{Binding FirstName}"/>
       <Label Text="Last name:"/>
       <Entry Text="{Binding LastName}"/>
       <Label Text="Date of birth:"/>
       <DatePicker Date="{Binding DateOfBirth,
           Mode=TwoWay}"/>
       <StackLayout Orientation="Horizontal">
```

```
            <Label Text="Is Family Member?:"
            VerticalOptions="Center"/>
            <CheckBox IsChecked="{Binding IsFamilyMember}"
            VerticalOptions="Center"/>
            </StackLayout>
          </StackLayout>
        </DataTemplate>
      </CollectionView.ItemTemplate>
    </CollectionView>
    <StackLayout Orientation="Horizontal">
      <Button Text="Add" Command="{Binding AddCommand}"/>
      <Button Text="Delete" Command="{Binding DeleteCommand}"
          CommandParameter="{Binding Source={x:Reference
          ContactsList},
          Path=SelectedItem}"/>
    </StackLayout>
  </StackLayout>
</ContentPage>
```

All the bound views point to objects defined in the viewmodel. As you can see, the **`CollectionView`** is still defined like in the dedicated example, and it still points to data via Binding expressions. At the bottom of the code, you can see how, for the two **`Button`** views, the **`Command`** property is assigned with a Binding expression that points to a command in the viewmodel. This way, no **`Click`** event is defined and handled in the view's code-behind, which perfectly satisfies the separation principles of MVVM. Note how, for the second button, a **`CommandParameter`** is also specified. This property allows for passing an object as a parameter to the bound **`Command`**. In this case, the parameter is the currently selected item (**`Path=SelectedItem`**) in the **`CollectionView (x:Reference ContactsList)`**. The last part involves assigning an instance of the viewmodel as the data source for the page, which can be done in the code-behind as follows:

```
private ContactViewModel ViewModel { get; set; }
public MainPage()
{
  InitializeComponent();
  ViewModel = new ContactViewModel();
  BindingContext = ViewModel;
}
```

If you run the code now, you will see a result similar to what you see in *Figure 10.9*:



First name:
Alessandro
Last name:
Del Sole
Date of birth:
5/10/1977
Is Family Member?: ☐
First name:
Robert
Last name:
White
Date of birth:
2/1/1960
Is Family Member?: ☐
First name:
ADD DELETE

Carrier 2:53 PM
First name:
Alessandro
Last name:
Del Sole
Date of birth:
5/10/77
Is Family Member?: ○
First name:
Robert
Last name:
White
Date of birth:
2/1/60
Is Family Member?: ○
First name:
Joseph
Add Delete

***Figure 9.10:*** *Data binding with Model-View-ViewModel*

Everything is working via data binding, and the viewmodel is working in an abstract way. Note how the **`Delete`** button is disabled until you select an item in the list; this happens because, behind the scenes, the view's **`IsEnabled`** property is automatically bound to the result of the **`CanExecute`** logic implemented for the command. As you can imagine, MVVM is much more than this, but what you have seen in this chapter is what you need to do to get started with this pattern and to be able to put your hands on existing MVVM implementations.

**A common task is opening another page from the current page, but in MVVM, this cannot be done directly (a view can only talk to its**

**viewmodel), and a viewmodel cannot open a view. There is a way to accomplish this via the `MessagingCenter` class, which is described in *Chapter 11, Managing the Application Lifecycle*, so you will still hear about MVVM.**

## MVVM frameworks

MVVM is a popular pattern, and many libraries have been created to make it simpler to work with, including support for much more complex tasks. The most popular ones are listed here:

- **FreshMvvm (http://www.michaelridland.com/xamarin/freshmvvm-mvvm-framework-designed-xamarin-forms/).**
- **MVVM Light** (**http://www.mvvmlight.net/**).
- **Prism** (**https://github.com/PrismLibrary/Prism**).
- **MvvmCross** (**https://www.mvvmcross.com/**).

FreshMvvm has been specifically designed for Xamarin.Forms, whereas Prism is backed by Microsoft. In general, they are all good libraries, and each one fully supports the MVVM pattern and principles.

## Pull-to-Refresh gesture: The RefreshView

Most applications allow reloading lists with a simple gesture. You keep a list pressed, move your finger down, and then release your finger for data to start reloading, and it shows an activity indicator. This gesture is known as pull-to-refresh. In Xamarin.Forms, the **`ListView`** is the only view with built-in pull-to-refresh capabilities, but it has not been discussed because there is a more recent and efficient way to implement pull-to-refresh in any view that supports scrolling, provided by the **`RefreshView`**. The **`RefreshView`** works with **`Command`** objects, and that is another reason why pull-to-refresh has not been discussed so far. Suppose you want to implement pull-to-refresh in the view defined in the latest example on MVVM. You can enclose the **`CollectionView`** inside a **`RefreshView`**, as follows:

```
<RefreshView RefreshColor="Blue"
    IsRefreshing="{Binding IsRefreshing}"
    Command="{Binding RefreshCommand}">
  <CollectionView x:Name="ContactsList"
    …
  </CollectionView>
</RefreshView>
```

There are three important properties that you assign:

- **`RefreshColor`**, of the **`Color`** type, allows you to assign a color to the activity indicator; otherwise, the system default is used.
- **`IsRefreshing`**, of the **`bool`** type, is assigned with true when you want to display the activity indicator and with false when you want to stop displaying the indicator.
- **`Command`** is assigned with a command that executes the action of reloading the bound data.

The **`IsRefreshing`** object is assigned with a binding expression, and the reason is that the data refreshing logic is inside the viewmodel, so you should not assign true or false directly in the view's code-behind. You will need to make a few changes to the viewmodel at this point. First, you need to add a property called **`IsRefreshing`**, as follows:

```
private bool _isRefreshing;
public bool IsRefreshing
{
  get
  {
   return _isRefreshing;
  }
  set
  {
   _isRefreshing = value;
   OnPropertyChanged();
  }
}
```

Second, you need a new command, as follows:

```
public Command RefreshCommand { get; set; }
```

In the constructor of the viewmodel, you then assign the command with some code that reloads the data. The following code provides an example that simulates a long-running operation:

```
RefreshCommand =
  new Command(async () =>
  {
   IsRefreshing = true;
   LoadSampleData();
   // Simulates a longer operation
   await Task.Delay(2000);
```

```
    IsRefreshing = false;
  }
 );
```

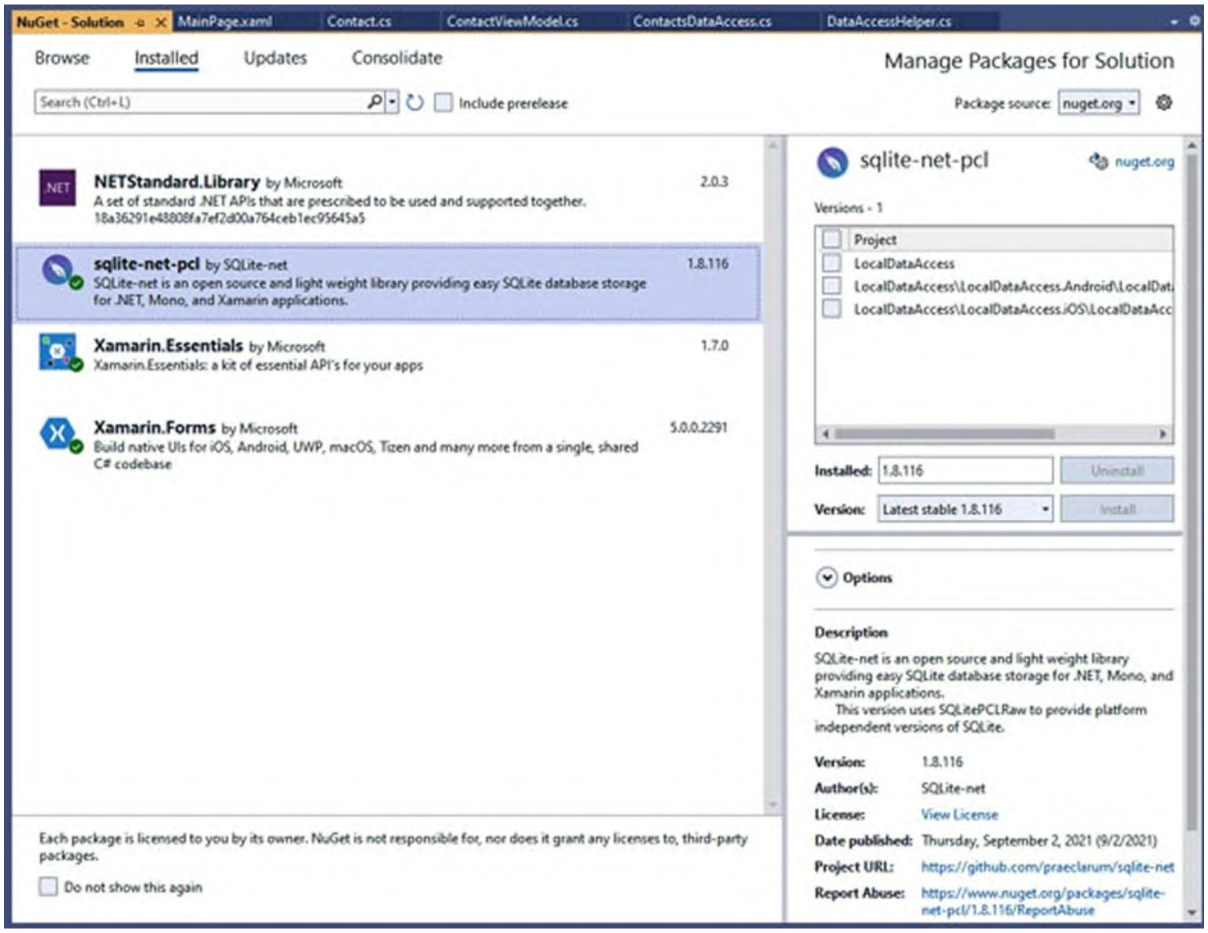
Data here is not coming from a database or a web service, so it would be immediately refreshed, and the **`Task.Delay`** method simulates waiting for some time. Note how the value for **`IsRefreshing`** is assigned before and after loading data and, when this happens, a change notification is raised so that the **`IsRefreshing`** property of the **`RefreshView`** is updated accordingly. You can now run the code again and try to do a pull-to-refresh gesture over the list of data.

## Local data access with SQLite databases

SQLite is a popular local database engine that ships pre-installed as part of both the Android and iOS operating systems. SQLite is a relational database that works with tables, columns, keys, indexes, and everything you would expect from a database. Additionally, it's perfect for storing application data in a structured way. SQLite databases are .db3 local files, which live inside the application's workspace on the device. This chapter explains how to work with SQLite databases in a real-world approach by implementing an appropriate architecture. It is recommended that you open the companion solution for this section, called **`LocalDataAccess`**, in Visual Studio. The topic is quite complex, and it will simplify the way you learn all the discussed areas. As an alternative, you can work on the MVVM sample solution, which is extended with the features discussed here.

## Installing the SQLite NuGet package

Before you write code, it is necessary to install a NuGet package that allows working against SQLite with C# and its advanced features, such as LINQ. A lot of libraries exist, but the one you need to install is called **`sqlite-net-pcl`** and is produced by *Praeclarum*, as you can see in *Figure 9.11*. This is a free library, also available as an open-source project:



NETStandard.Library by Microsoft 2.0.3
A set of standard .NET APIs that are prescribed to be used and supported together.
18a36291e48808fa7ef2d00a764ceb1ec95645a5

sqlite-net-pcl by SQLite-net 1.8.116
SQLite-net is an open source and light weight library providing easy SQLite database storage for .NET, Mono, and Xamarin applications.

Xamarin.Essentials by Microsoft 1.7.0
Xamarin.Essentials: a kit of essential API's for your apps

Xamarin.Forms by Microsoft 5.0.0.2291
Build native UIs for iOS, Android, UWP, macOS, Tizen and many more from a single, shared C# codebase

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

sqlite-net-pcl nuget.org

Versions - 1

| | Project |
|---|---|
| ☐ | LocalDataAccess |
| ☐ | LocalDataAccess\LocalDataAccess.Android\LocalDat |
| ☐ | LocalDataAccess\LocalDataAccess.iOS\LocalDataAcc |

Installed: 1.8.116 Uninstall

Version: Latest stable 1.8.116 Install

Description

SQLite-net is an open source and light weight library providing easy SQLite database storage for .NET, Mono, and Xamarin applications.
This version uses SQLitePCLRaw to provide platform independent versions of SQLite.

Version: 1.8.116
Author(s): SQLite-net
License: View License
Date published: Thursday, September 2, 2021 (9/2/2021)
Project URL: https://github.com/praeclarum/sqlite-net
Report Abuse: https://www.nuget.org/packages/sqlite-net-pcl/1.8.116/ReportAbuse

***Figure 9.11:*** *Installing the SQLite NuGet Packages*

Make sure the package is installed for all the projects in the solution.

## Getting the database path

Once the NuGet package has been installed, it is a good idea to define a class with a static property that contains the path of the database file on disk. In the companion solution, you can find a file called **`DataAccess\DataAccessHelper.cs`**, whose code is as follows:

```
using System;
using System.IO;
namespace LocalDataAccess.DataAccess
{
  internal class DataAccessHelper
  {
   public const string DatabaseFilename = “Contacts.db3";
   public static string DatabasePath
   {
    get
```

```
      {
       var basePath = Environment.
         GetFolderPath(Environment.SpecialFolder.
         LocalApplicationData);
       return Path.Combine(basePath, DatabaseFilename);
      }
    }
  }
}
```

The core part of this code is the **DatabasePath** property, of the **string** type, which returns the path for the database file. It is made of the local application folder and of the database filename, identified by the **DatabaseFilename** field. The folder is retrieved by a method called **GetFolderPath**, from the static **Environment** class, and is identified via the **LocalApplicationData** variable from the **Environment.SpecialFolder** enumeration. This code allows for retrieving the appropriate path, regardless of the target system; the path will be used shortly.

## Implementing a data model

The goal of the current example is to read from and write to a database collection of **Contact** objects. To accomplish this, the existing **Contact** class is used as a data model to map a table in the database, but it needs a few adjustments. First, you need to add a using SQLite; directive to the **Contact.cs** file, and the class definition should be changed as follows:

```
[Table(“Contacts")]
public class Contact : INotifyPropertyChanged
```

If the **Table** attribute was not specified, the table on the database would be created using the class name, so the table would be called **Contact**. By convention, table names are usually pluralized, so the **Table** attribute is used for this purpose, but in general, it allows for providing a table name that is different from the class name. The second edit you need to do is add an integer property that is used by the database as a primary key so that objects can be indexed and identified quickly. Having that said, you could add the following property:

```
[PrimaryKey][AutoIncrement]
public int ID { get; set; }
```

Note how the **PrimaryKey** and **AutoIncrement** attributes are applied to the property so that it is used as primary key and its value is automatically incremented and handled by the database engine. These changes could be enough, but it is also very useful for you to know that SQLite supports data validation by adding special attributes to data properties. For example, you could add the following attributes to the **`LastName`** property:

```
[MaxLength(50)][NotNull]
public string LastName
```

**`MaxLength`** determines the maximum length for the string, and **`NotNull`** makes it mandatory to provide a value for this property. Now that you have made all the minimum necessary changes, it is time to implement a data access layer.

## Implementing a data access layer

In the real world, it is a best practice to implement a data access layer, typically, a class that exposes methods that perform operations against data and that is independent of all other parts of the applications. For better understanding, in the companion solution, consider the **`DataAccess\ContactsDataAccess.cs`** file. The first part of the code defines an object of the **`SQLiteConnection`** type, whose instance allows for accessing the database and for performing operations against data:

```
public class ContactsDataAccess
{
  public static SQLiteConnection Database;
  private static object collisionLock = new object();
  public ContactsDataAccess()
  {
    Database = new
    SQLiteConnection(DataAccessHelper.DatabasePath);
    Database.CreateTable<Contact>();
  }
```

The **`collisionLock`** field, of the object type, will be shortly used to prevent multiple threads from accessing the database. The constructor of the **`SQLiteConnection`** class takes the database path as an argument that is exposed by the previously defined **`DataAccessHelper.DatabasePath`** property.

**Tip: If a database with the specified name and path is not found, the SQLite engine creates a new one.**

The **`CreateTable`** method is required to create a table the first time the database is created, but if the table already exists, this call will simply be ignored. Such a method requires you to specify the class that maps the desired table, in this case, the **`Contact`** class that maps a **`Contacts`** table. The **`SQLiteConnection`** class also exposes methods to read and write data. The **`Table`** method returns a collection of objects if the table exists, and it creates one if it is the first time the database is accessed. The benefit of implementing a data access layer is that you can implement further logic. For example, the following method returns only the list of contacts who are also family members:

```
public List<Contact> GetFamilyMembers()
{
 lock (collisionLock)
 {
  var contacts = Database.Table<Contact>();
  var result = contacts.Where(c => c.IsFamilyMember).ToList();
  return result;
 }
}
```

As you can see, you can use LINQ and filter a collection based on your business requirements. Note how the code that accesses the database is encapsulated inside a **`lock`** block. This prevents other threads from accessing the database when it is already in use. **`Table`** is a generic method and takes the class of interest as the **`type`** parameter. This is how the method can understand which table you want to get the data from. **`SQLiteConnection`** exposes methods that match the so-called **C.R.U.D** (**Create, Read, Update, Delete**) operations. They are **`Insert`**, **`Update`**, **`Delete`**, **`InsertAll`**, **`UpdateAll`**, **`DeleteAll`**. The first three methods work against an individual instance of an object, whereas the other methods perform the related operation against all the items in the specified collection. The following code demonstrates how to insert, edit, and delete a contact:

```
public void AddContact(Contact contact)
{
 lock (collisionLock)
 {
  Database.Insert(contact);
 }
}
public void DeleteContact(Contact contact)
{
 lock(collisionLock)
 {
   Database.Delete(contact);
  }
 }
public void EditContact(Contact contact)
{
 lock(collisionLock)
 {
  Database.Update(contact);
 }
}
```

The usage of these methods is very easy. These methods could include further logic before data is processed, if required. Now, suppose you have new but unsaved contacts in a collection, and that you have made changes to the existing contacts. The following code demonstrates how to work with multiple objects:

```
public void SaveAll(IEnumerable<Contact> contacts)
{
 lock(collisionLock)
 {
  var existingContacts = contacts.Where(c => c.ID != 0);
  var newContacts = contacts.Where(c => c.ID == 0);
  Database.UpdateAll(existingContacts);
  Database.InsertAll(newContacts);
 }
}
```

The **`SaveAll`** method receives the full collection of objects you want to manipulate, and it will be sent by the viewmodel. The key point of this code is that value of the primary key, of the **`int`** type, is zero until the object is not saved to the database, so you can distinguish between unsaved objects that need to be inserted, and existing objects whose value is not zero and that just need to be updated. You could implement additional queries and logic, but these methods are enough for the example's purposes. It is worth mentioning that the **`SQLiteConnection`** class also exposes a method called **`Query`**, which allows the execution of SQL statements directly instead of using LINQ.

## Invoking the data access layer

The data access layer must be invoked by the viewmodel, which is an intermediate object between the data and the user interface. The **`ContactViewModel`** class should first be extended with the following two properties:

```
public Command SaveAllCommand { get; set; }
public ContactsDataAccess ContactsDataBase;
```

The first command adds the option to save all the objects in the collection based on the **`SaveAll`** method described previously. The **`ContactsDataBase`** property, of the **`ContactsDataAccess`** type, represents an instance of the data access layer class. Next, you can define a method that loads data. In the previous examples of these chapters, data was created manually, but now it is loaded from the database. The method is called **`LoadData`** and looks as follows:

```
private void LoadData()
{
  Contacts = new ObservableCollection<Contact>
    (ContactsDataBase.GetFamilyMembers());
}
```

Note how the **`GetFamilyMembers`** method is called to retrieve only instances of the **`Contact`** class whose **`IsFamilyMember`** property is true, and how the result is translated into an **`ObservableCollection<Contact>`** and assigned to the **`Contacts`** property of the viewmodel, which is the real data source. The final step required to make the viewmodel work against data involves creating an instance of the data access layer class and implementing the new command. You can edit the constructor of the viewmodel as follows, adding the pieces highlighted in bold:

```
public ContactViewModel()
{
  ContactsDataBase = new ContactsDataAccess();
  LoadData();
  AddCommand =
    new Command(() => Contacts.Add(new Contact()));
  DeleteCommand =
    new Command<Contact>((contact) =>
    {
      Contacts.Remove(contact);
      if(contact.ID != 0)
      ContactsDataBase.DeleteContact(contact);
    },
    (contact) => contact.LastName != null);
  SaveAllCommand = new Command(() =>
  ContactsDataBase.SaveAll(Contacts));
  RefreshCommand =
    new Command(async () =>
    {
      IsRefreshing = true;
      LoadData();
      // Simulates a longer operation
      await Task.Delay(2000);
      IsRefreshing = false;
    }
  );
}
```

The following is a list of key points:

- An instance of the data access layer (**`ContactsDataAcces`** class) is created, and the **`LoadData`** method is invoked.
- The **`AddCommand`** implementation does not change because a new object only needs to be added in memory.
- The **`DeleteCommand`** implementation is extended to also delete the object from the database, but only if its ID is not 0 (which means it already exists in the table), via the **`DeleteContact`** method of the data access layer class.
- The new **`SaveAllCommand`** object simply invokes the **`SaveAll`** method of the data access layer, passing the current collection of contacts. The latter method is responsible for performing the logic over new and existing items.

Now, it is time to finalize the project by extending the user interface, and some of the benefits of MVVM will be even clearer.

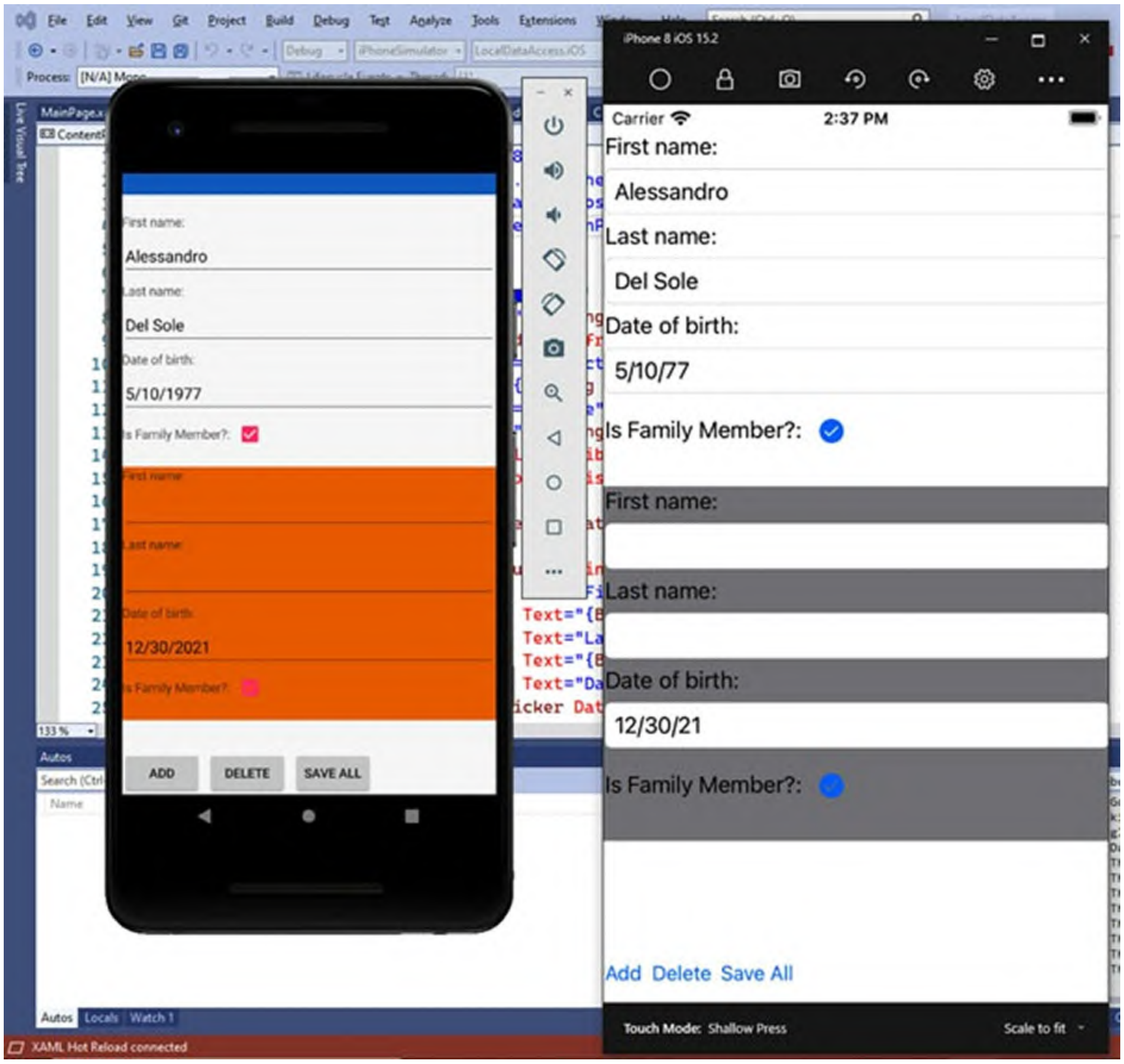
## Extending the user interface

All the changes you have made to load and save data working against a database have been done in the viewmodel, inside the commands' implementation, so nothing needs to be changed in the user interface. In fact, because of data binding and MVVM principles, the user interface is bound to commands in the viewmodel without knowing what actually happens behind the scenes, and this is one of the major benefits of MVVM. The only thing you need to do is add a new button bound to the **`SaveAllCommand`** property, which you can add as follows:

```
<Button Text="Save All" Command="{Binding SaveAllCommand}"/>
```

If you now run the application, you will first see an empty list that you can start populating by clicking on the **`Add`** button. You will also see how the **`Delete`** button is disabled until a contact is selected in the list (see *Figure*

9.12), and this is because you have specified the **`CanExecute`** parameter in the command implementation:


First name:
Alessandro
Last name:
Del Sole
Date of birth:
5/10/1977
Is Family Member?: ✓

First name:
Last name:
Date of birth:
12/30/2021
Is Family Member?:

First name:
Alessandro
Last name:
Del Sole
Date of birth:
5/10/77
Is Family Member?: ✓

First name:
Last name:
Date of birth:
12/30/21
Is Family Member?:

***Figure 9.12:*** *Working with a local SQLite database*

Clicking on **`Save All`** will allow you to see how all new and edited **`Contact`** instances will be saved into table rows. Obviously, there are infinite possibilities of querying data and of performing business logic over your objects, but it all depends on the business needs. You now have all the necessary knowledge to be immediately productive on projects that implement data binding and local data access, even with complex architectures like MVVM.

## Conclusion

Resources allows you to reuse styles, data templates, and converters, whereas data binding allows you to automate the communication between data and the user interface. By implementing patterns like Model-View-ViewModel, you have complete separation of layers, and you can change the behavior and logic without affecting the user interface and vice versa. Switching from sample data supplied in code to a local SQLite database has been a good example of the benefits of MVVM. Now, you are aware of most of what Xamarin.Forms offers in terms of development features and techniques. In the next chapter, you will extend your knowledge and learning about graphics and media.

## Key terms

- **Resource**: A reusable set of XAML properties, like styles and data-templates.
- **Style**: A resource that makes it easy to apply the same properties to views of the same type.
- **Data-template**: A resource that determines how an object in a list should be presented.
- **Data binding**: Automatic information exchange between views and .NET objects.
- **Model-View-ViewModel**: An architectural pattern that allows for clear separation between layers, such as data (`Model`), business logic (`ViewModel`), and user interface (`View`).

# CHAPTER 10
# Brushes, Shapes, and Media

## Introduction

Mobile applications should not just be functional; they should also provide a beautiful user interface. This might seem like an annoying sentence because obviously, serving the purpose in the most efficient way should certainly be the focus of any app. However, the more an app is appealing, the more a user will likely return and use it repeatedly, which is key for every app producer. The better the user experience and the user interface, the more users will be attracted by an app before they get on board with the features. There are many ways to make beautiful apps in Xamarin.Forms, and this chapter focuses on recent additions to the code base: brushes, shapes, and support for multimedia contents. You will learn how to use these objects from a drawing perspective because you need to know how they technically work. However, you will also be provided with hints about possible usage in the real-world to improve the quality of the user interface.

## Structure

In this chapter, we will cover the following topics:

- Coloring objects with brushes
- Drawing shapes
- Working with multimedia

## Objectives

By completing this chapter, you will be able to add solid colors, gradients, geometrical and custom shapes, and media playing features to your applications.

## Creating a sample project

This chapter comes with a companion Xamarin.Forms solution that you can open with Visual Studio to better understand the examples. If you wish to create a project on your own from scratch, you can follow these steps:

1. Create a new Xamarin.Forms solution called **`Brushes_Shapes_Media`** for consistency with the sample solution.
2. Do not edit or remove the auto generated **`MainPage.xaml`** file; it will be used later.
3. For each layout discussed in the book, add a new item of type **`Content Page (XAML)`**. To accomplish this, right-click on the shared project name and then click on **`Add New Item`** in the **`Context`** menu.
4. In the **`Add New Item`** dialog, click on the Xamarin.Forms node on the left and then select the **`Content Page (XAML)`** item template.
5. Assign to the new XAML file a name that matches the discussed view, for example, **`Brushes.xaml`**, and click on **`Add`**.

For each page you add to the project, add an empty **`StackLayout`** to the **`ContentPage`** and assign its **`VerticalOptions`** property with **`CenterAndExpand`**. Unless specified, this will be the layout of choice for the code examples in the upcoming segments.

## Coloring objects with brushes

Brushes are a recent addition to the Xamarin.Forms code base. They are objects that derive from the **`Brush`** base class and that allow for coloring specific views with solid colors and gradients. For a better understanding, consider the following Frame definition, where the **`BackgroundColor`** property is assigned the normal way:

```
<Frame CornerRadius="5" BackgroundColor="Yellow"
    Margin="20" WidthRequest="150"
    HeightRequest="150">
</Frame>
```

The **`BackgroundColor`** property supports values of type **`Color`**. Most views now also expose a property called **`Background`**, which is of type **`Brush`**. You could rewrite the previous code as follows:

```
<Frame CornerRadius="5" Margin="20" WidthRequest="150"
HeightRequest="150">
```

```
  <Frame.Background>
   <SolidColorBrush Color="Yellow"/>
  </Frame.Background>
</Frame>
```

Properties of type **`Brush`** allow for assigning objects of type **`SolidColorBrush`**, **`LinearGradientBrush`**, and **`RadialGradientBrush`**.

**The purpose of brushes is to fill a view's background, so this is the reason why `Background` is the property of type `Brush` you work the most with.**

A **`SolidColorBrush`** object fills a view with an individual color, and you assign its **`Color`** property (of type **`Color`**) with the color of interest. When working with individual colors, you can also use the following, compact syntax:

```
<Frame CornerRadius="5" Background="Yellow" Margin="20"
WidthRequest="150"
   HeightRequest="150">
</Frame>
```

You can also create linear gradients, both vertical and horizontal, with the **`LinearGradientBrush`** objects and circular gradients with the **`RadialGradientBrush`** object.

**Tip: The only reason for using an individual color by assigning a property of type Brush instead of a property of type `Color` is that you can replace the brush at runtime with a different one, such as gradients. Otherwise, the choice would make no difference.**

## Defining linear gradients

The second brush to mention is the **`LinearGradientBrush`**. This allows for creating horizontal, vertical, and diagonal gradients. Linear gradients are drawn based on two-dimensional coordinates, called **`StartPoint`** and **`EndPoint`**, respectively. **`StartPoint`** represents the top-left corner of the view, and the default value is 0,0 (which is also its minimum). **`EndPoint`** represents the view's bottom-right corner, and the default value is 1,1 (which is also its maximum). In the following code, you can see how to fill the background of a **`Frame`** with a horizontal linear gradient:

```
<Frame CornerRadius="5" Margin="20" WidthRequest="150"
```

```
HeightRequest="150">
  <Frame.Background>
    <LinearGradientBrush EndPoint="1,0">
      <GradientStop Color="Blue" Offset="0.1" />
      <GradientStop Color="Violet" Offset="0.5" />
      <GradientStop Color="Red" Offset="1.0" />
    </LinearGradientBrush>
  </Frame.Background>
</Frame>
```

As you can see, each color in the gradient is set with a **`GradientStop`** object, which also represents the position of the color in the gradient (**`OffSet`** property, of type double). There is no limit to the number of colors you can add. You control the direction of the gradient via the **`StartPoint`** and **`EndPoint`** values. *Figure 10.1* shows the result of this code:

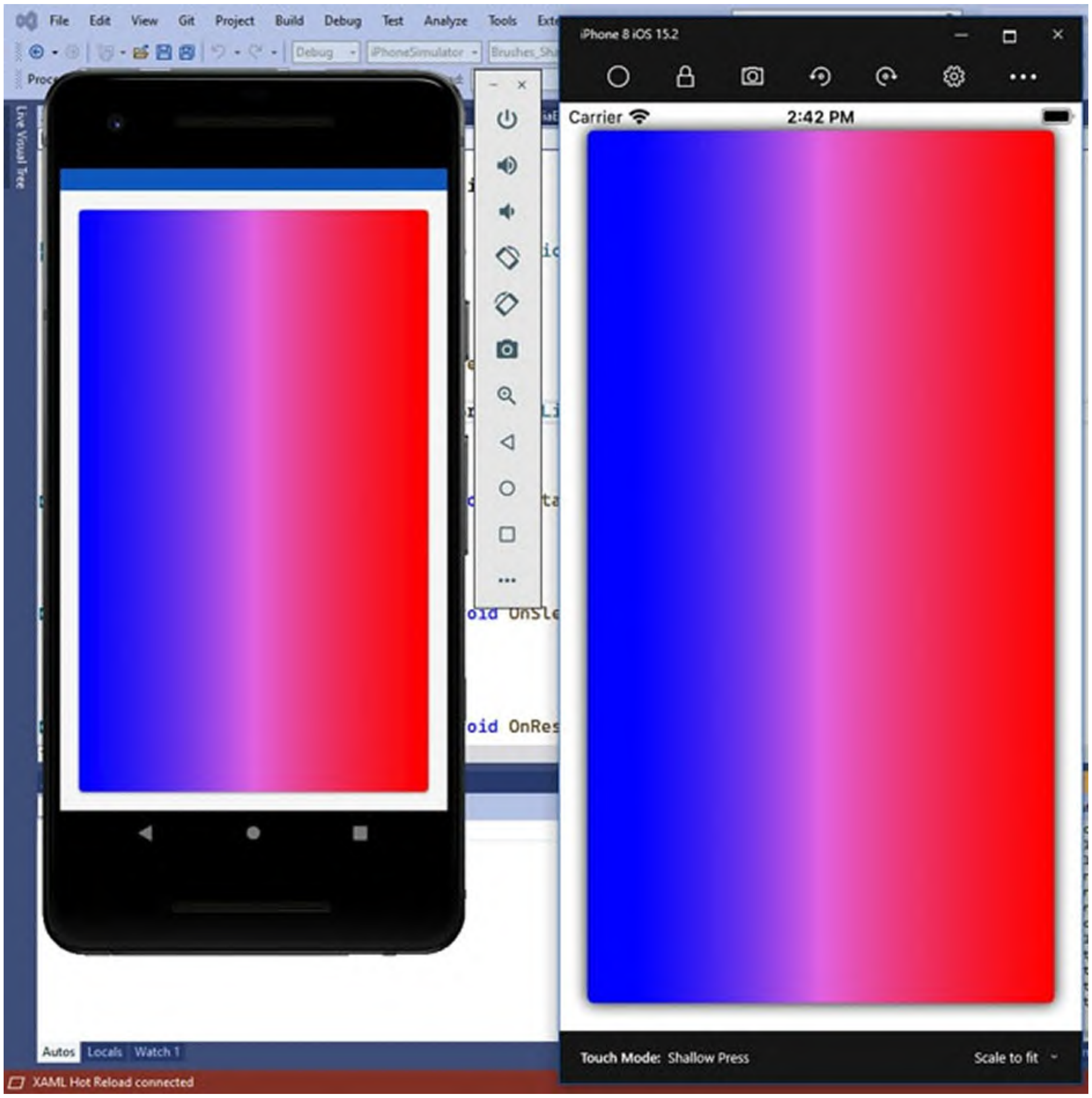


***Figure 10.1:*** *Adding a linear gradient*

Given their values, if you do not specify **`StartPoint`** and **`EndPoint`**, the linear gradient will be drawn diagonally, from top-left to bottom-right.

## Defining circular gradients

The last available brush is called **`RadialGradientBrush`** and allows for filling views with a circular gradient. It still works with **`GradientStop`** objects like the **`LinearGradientBrush`** to represent colors and their position, and it exposes two new properties: **`Radius`**, of type double, which determines the radius of the circle, with a default value of **`0.5`** in a range of 0 to 1; and **`Center`**, of type **`Point`**, whose default value is **`0.5`**, **`0.5`** and that represents the center of the circle for the gradient. The following code shows an example:

```
<Frame CornerRadius="5" Margin="20" WidthRequest="150"
HeightRequest="150">
  <Frame.Background>
    <RadialGradientBrush>
      <GradientStop Color="Blue"
        Offset="0.1" />
      <GradientStop Color="Violet"
        Offset="0.5" />
      <GradientStop Color="Red"
        Offset="1.0" />
    </RadialGradientBrush>
  </Frame.Background>
</Frame>
```

The result for this code is shown in the *Figure 10.2*:

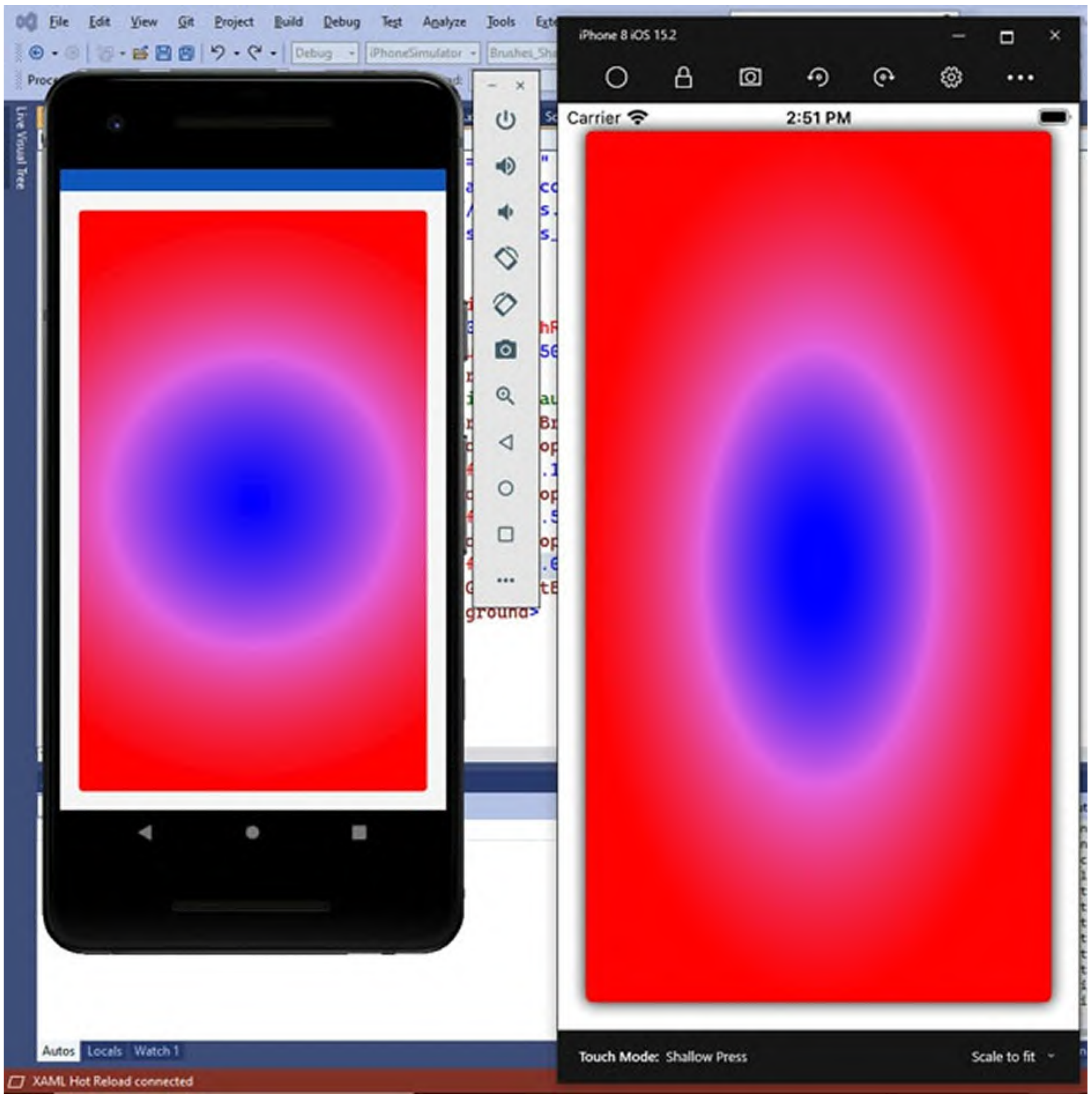

File Edit View Git Project Build Debug Test Analyze Tools
iPhone 8 iOS 15.2
Carrier
2:51 PM
Touch Mode: Shallow Press
Scale to fit
Autos Locals Watch 1
XAML Hot Reload connected

***Figure 10.2:*** *Adding a circular gradient*

In the real-world, you will likely use linear gradients more than radial gradients, but it is always good to know about this possibility.

**As you can see in *figure 10.2*, the circle is not appearing the same on Android and iOS. This is one of those cases where the look of a view depends on each system, screen size and form factor.**

## Drawing shapes

Shapes are special views that allow you to draw geometries in your pages that derive from the base **Shape** class. **Shape** objects in Xamarin.Forms can render both regular and custom shapes. When talking about regular shapes, the following are available: **Ellipse**, **Rectangle**, **Line**, **Polygon**. As you can see, they all have self-explanatory names. When it comes to custom shapes, you will use the **Polyline** object. Before going into the details of each shape, it is worth mentioning that they share several properties, summarized in *Table 10.1*:

| Property | Type | Description |
|---|---|---|
| **Aspect** | **Stretch** | Determines how the shape fills the surrounding space. Supported values are `None`, `Fill`, `Uniform`, and `UniformToFill`. |
| **Fill** | **Brush** | The brush used to fill the shape. |
| **Stroke** | **Brush** | The brush used to draw the shape's outline. |
| **StrokeThickness** | **double** | Represents the width of the outline. The default is 0, which means an invisible outline. |
| **StrokeDashArray** | **DoubleCollection** | A collection of double values that represent the pattern used to draw dashes and gaps for a shape's outline. |
| **StrokeDashOffset** | **double** | Represents the distance between dashes. |

***Table 10.1:*** *Common shape properties*

While working with shapes, you will see how easy it is to learn about these properties. It is worth mentioning that, for the sake of design clarity, an outline will always be added to all the code examples, but this is completely optional.

## Drawing circles and ellipses

The simplest shape available is the **Ellipse**, which can be used to draw circles and ellipses. The following code demonstrates how to draw a simple ellipse:

```
<Ellipse Fill="Yellow" Stroke="Green" StrokeThickness="3"
WidthRequest="250"
   HeightRequest="100" HorizontalOptions="Center"
   Margin="0,50,0,0"/>
```

The **Fill** property contains a **Brush** object that fills the shape, so you are not limited to an individual color. The same holds for the **`Stroke`** property, which is the brush used to draw the outline. You can control the size of the ellipse via its **`WidthRequest`** and **`HeightRequest`**. When they have the same value, the ellipse will appear as a circle. *Figure 10.3* shows how the ellipse appears:

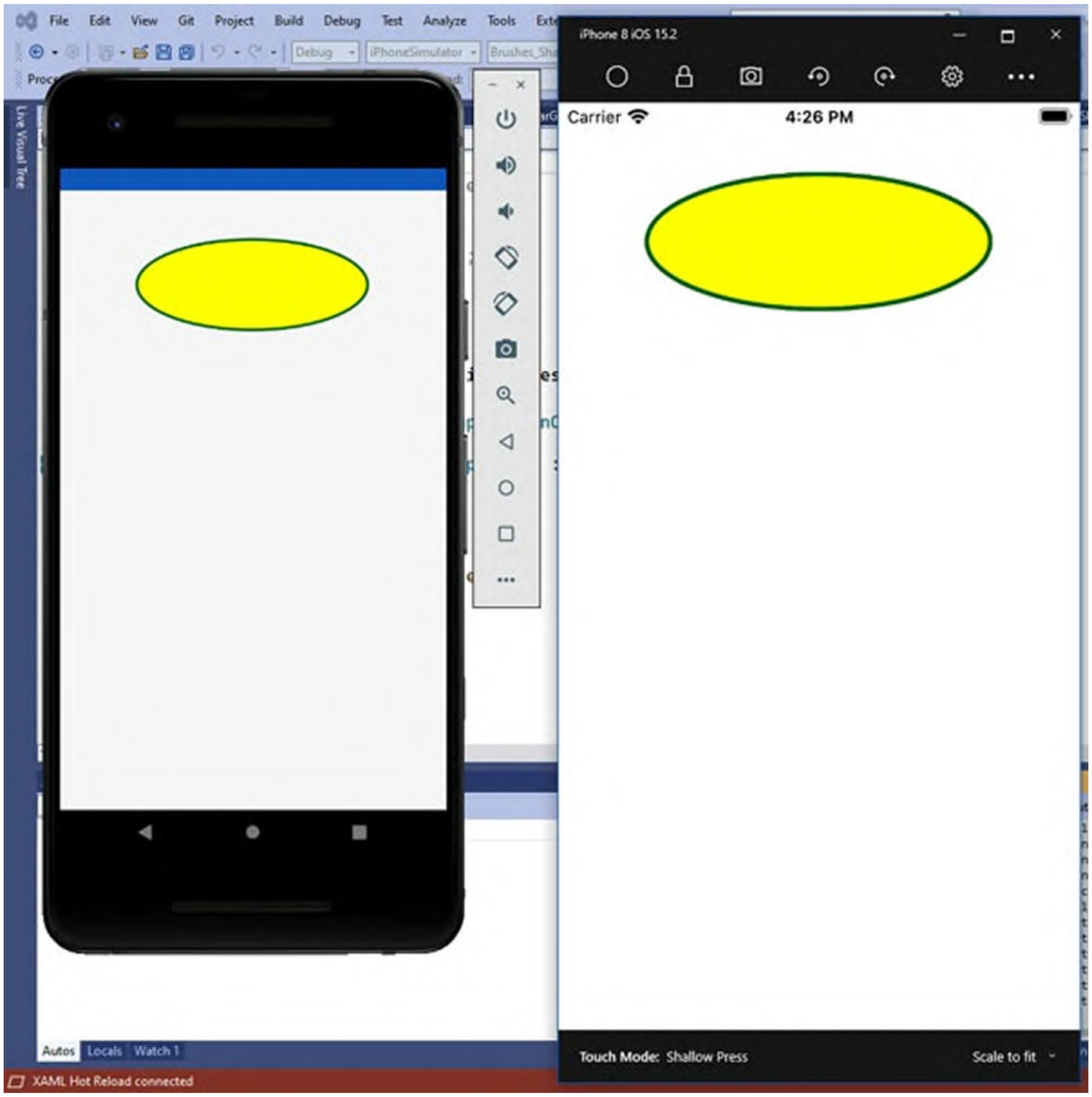


***Figure 10.3:*** *Drawing an ellipse*

One useful way to take advantage of the **`Ellipse`** in the real world could be creating avatars, with **`Ellipse`** views filled in with Image views, such as for profile pictures.

## Drawing rectangles

The next shape is another easy one: the **Rectangle**. The following code demonstrates how to draw a rectangle:

```
<Rectangle Stroke="Green" StrokeThickness="4"
StrokeDashArray="1,1"
StrokeDashOffset="6" WidthRequest="250" HeightRequest="100"
Margin="0,50,0,0"
HorizontalOptions="Center">
  <Rectangle.Fill>
    <LinearGradientBrush>
      <GradientStop Color="Yellow" Offset="0"/>
      <GradientStop Color="Red" Offset="0.5"/>
      <GradientStop Color="Orange" Offset="1"/>
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

If you look back at *Table 10.1*, you can recall how the **StrokeThickness** property determines how thick the outline is, whereas **StrokeDashArray** and **StrokeDashOffset** represent the pattern and distance of dashes, respectively. *Figure 10.4* shows the resulting shape:

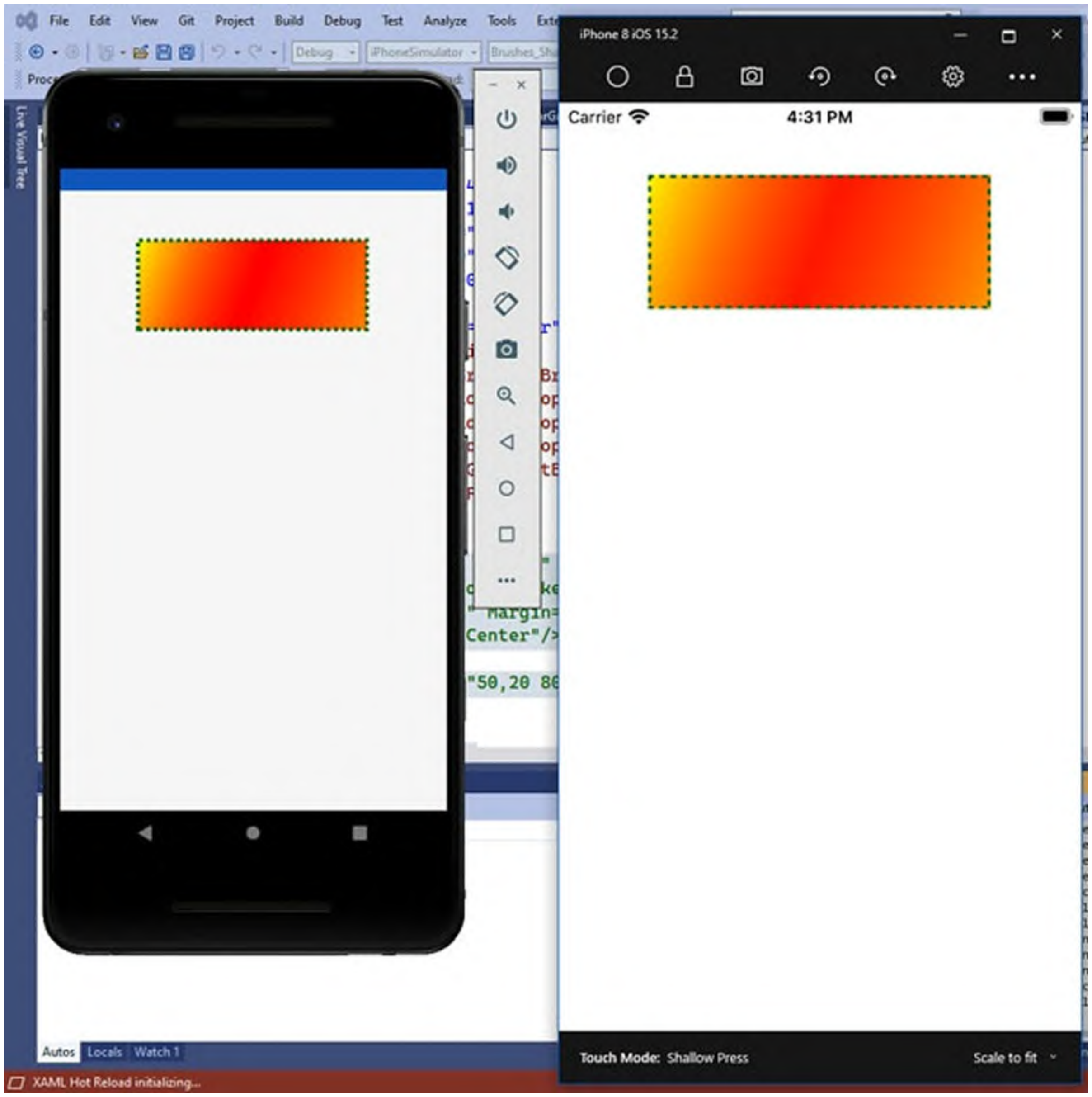


***Figure 10.4:*** *Drawing a rectangle*

## Drawing lines

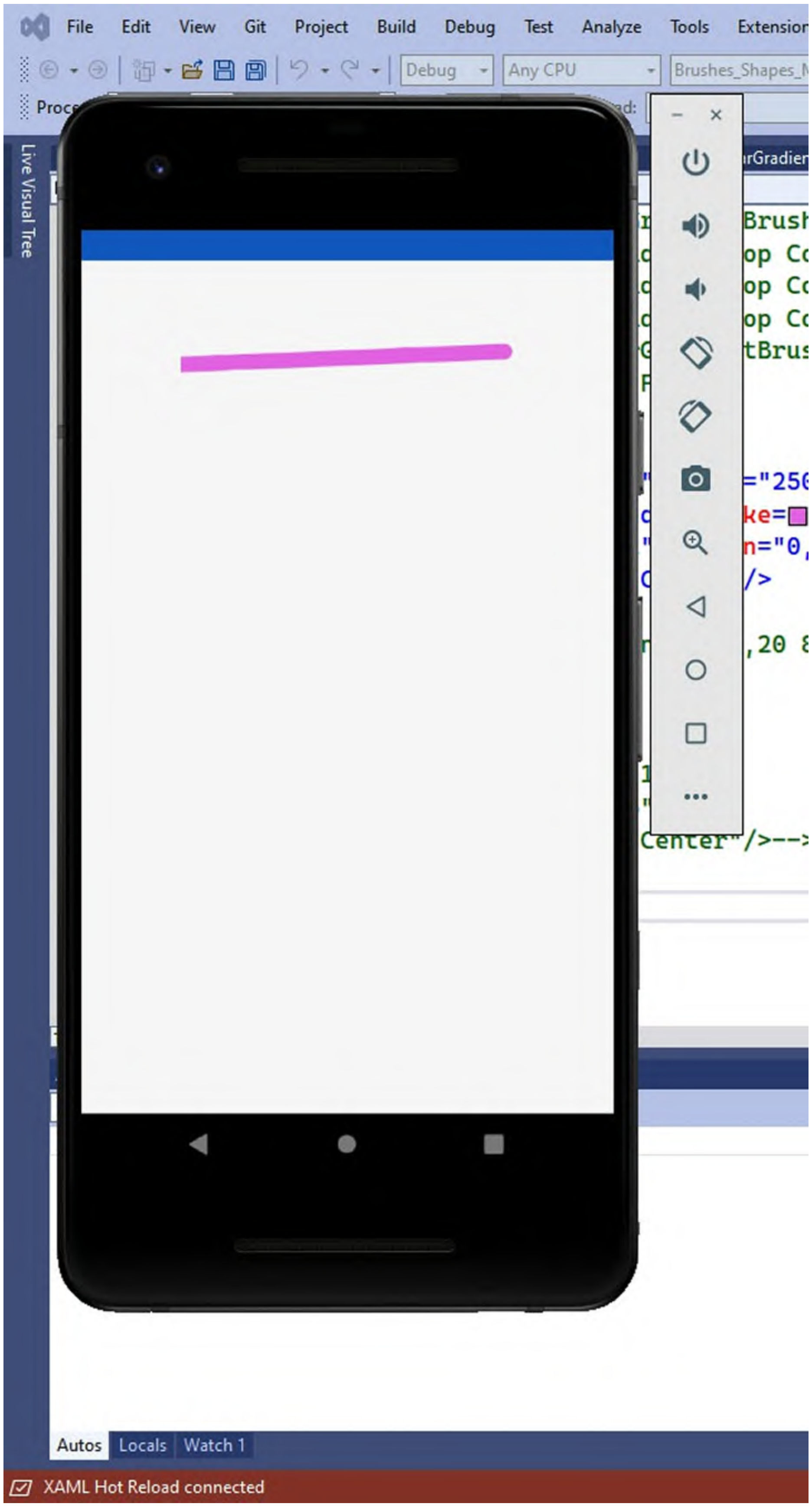
It is very easy to draw lines with the Line shape. Consider the following code:

```
<Line X1="0" Y1="30" X2="250" Y2="20" StrokeLineCap="Round"
Stroke="Violet"
    StrokeThickness="12" Margin="0,50,0,0"
    HorizontalOptions="Center"/>
```

The relevant properties are **`X1`**, **`Y1`**, **`X2`**, and **`Y2`**, all of type double. **`X`**1 and **`X2`** represent the starting and ending points of the line on the X axis, whereas **`Y1`** and **`Y2`** represent the lowest and highest points of the line on the Y axis. The **`StrokeLineCap`** property, which is totally optional, allows for adding a shape at the end of the line. You can choose one between **`Flat`** (default, no shape), **`Square`** (rectangle with the same thickness and height of the line), and **`Round`** (semicircle with a diameter equal to the thickness of the line).

**There is a known bug in the latest version of Xamarin.Forms that causes an app to crash on iOS when rendering a Line object. Therefore, *figure 10.5* only shows the Android version. You can follow the development of the issue on GitHub at https://github.com/xamarin/Xamarin.Forms/issues/14986.**

*Figure 10.5* shows the result of the code:



XAML Hot Reload connected

***Figure 10.5:*** *Drawing a line*

## Drawing polygons

Polygons are generically complex shapes, and Xamarin.Forms offers the **`Polygon`** object to draw them. The most important property of a **`Polygon`** is **`Points`**, which is a collection of **`Point`** objects. Each **`Point`** represents the coordinate of a delimiter in the polygon. The following code demonstrates how to draw a triangle, with the delimiter coordinates specified in the **`Points`** property:

```
<Polygon Points="50,20 80,60 20,60" Fill="Yellow" Stroke="Green"
StrokeThickness="4" StrokeDashArray="1,1" StrokeDashOffset="6"
VerticalOptions="Center" HorizontalOptions="Center"/>
```

*Figure 10.6* shows the result of this code:

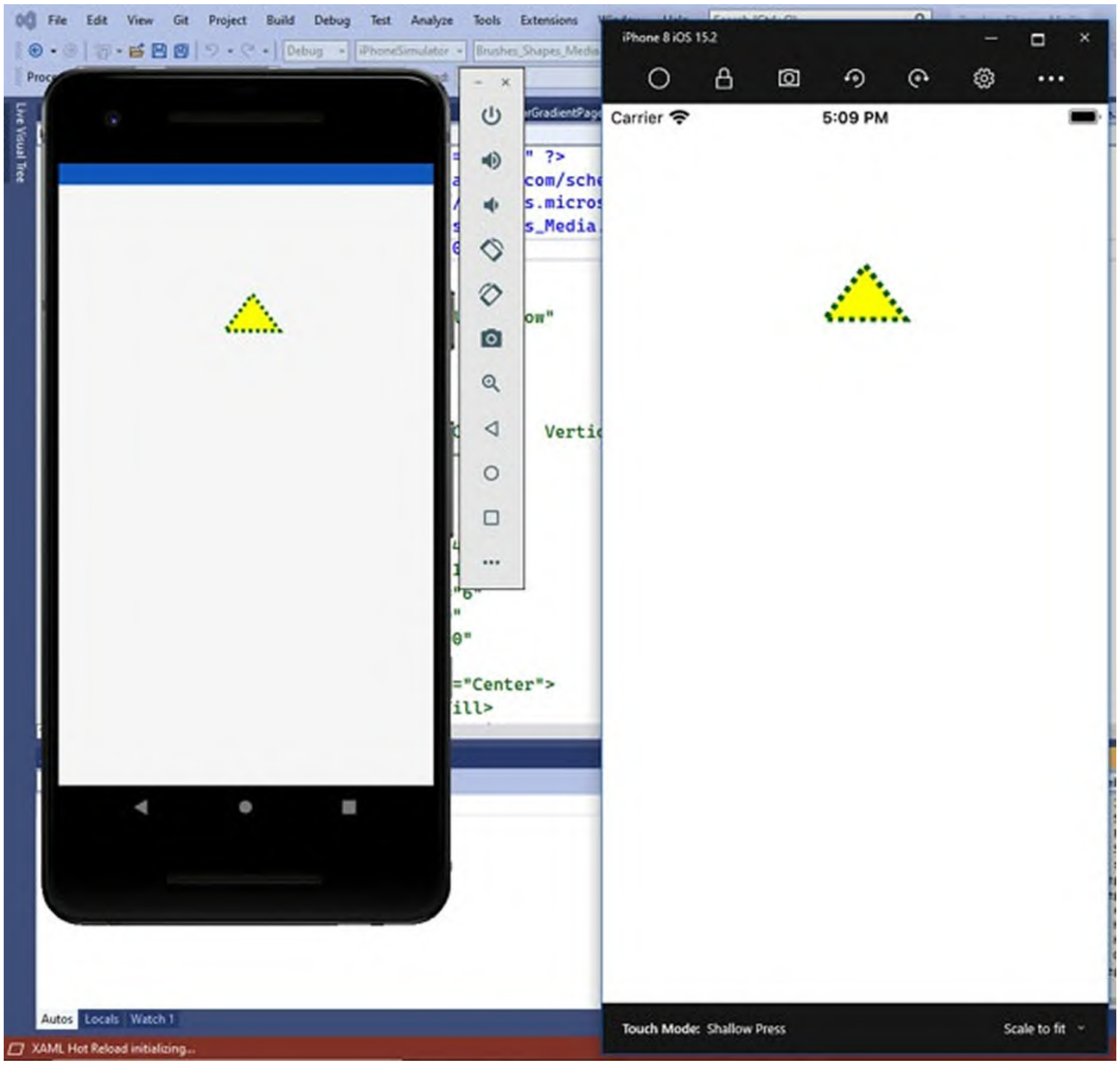


***Figure 10.6:*** *Drawing a polygon*

Note how you still use the other properties described earlier to draw and control the outline. There is no limit to the number of coordinates, so you can create complex polygons.

## Drawing custom shapes

Xamarin.Forms gives you great flexibility over shapes and allows you to create custom shapes. More specifically, the **`Polyline`** is a special type of shape made of several lines, all interconnected with one another, but the last line does not connect to the first point of the shape. For simpler understanding, consider the following code:

```
<Polyline Margin="0,50,0,0" HorizontalOptions="Center"
  Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48
  192, 150 200 144 48"
  Fill="Yellow" Stroke="Red" StrokeThickness="3"/>
```

As you can see, the way you define a **`Polyline`** is very close to the way you define a **`Polygon`**, since you still set its **`Points`** property with a collection of coordinates. However, the last line of the shape does not connect with its first point. This is demonstrated in *Figure 10.7*:

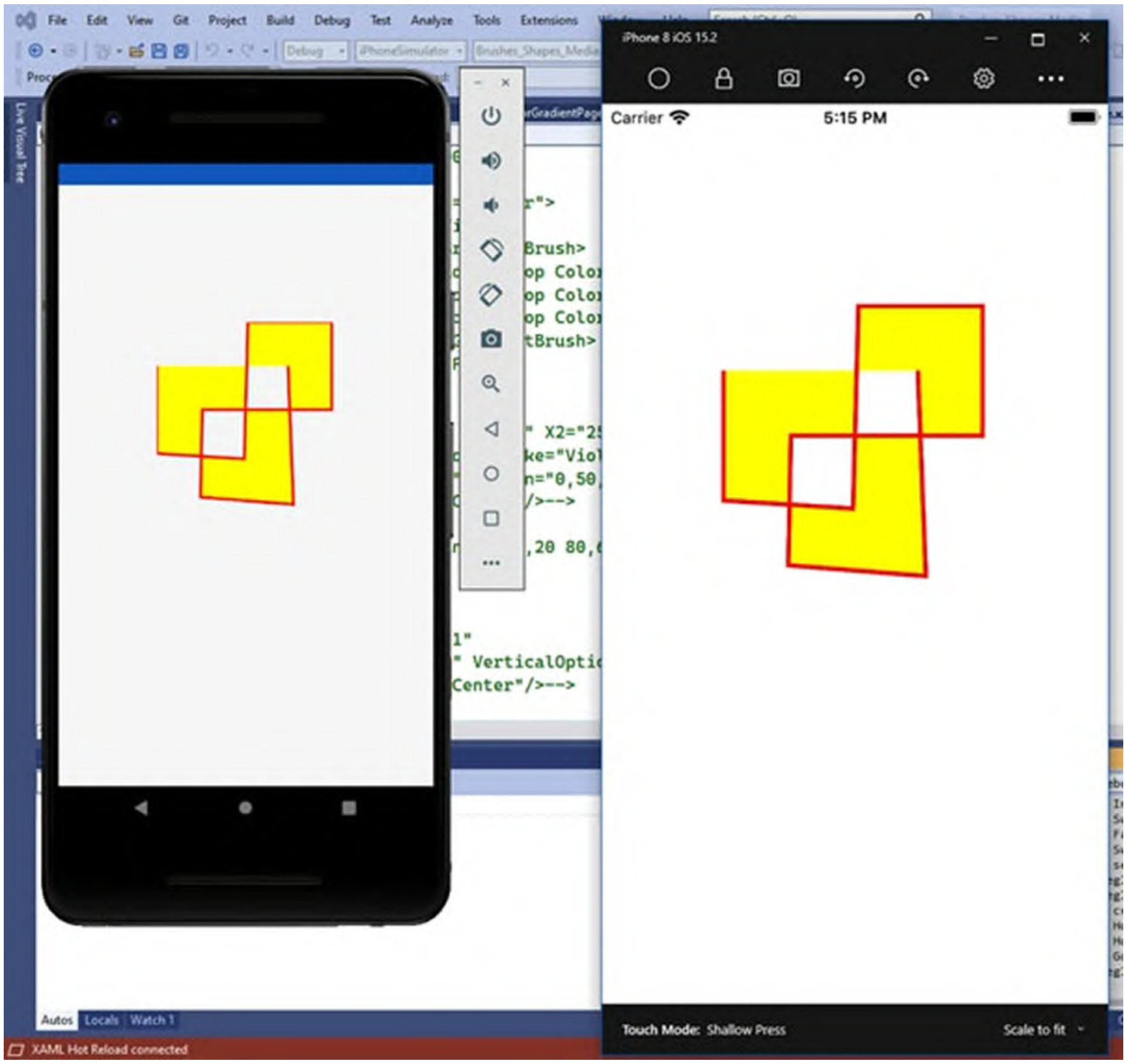


***Figure 10.7:*** *Drawing a complex shape with the Polyline*

Focus on the red lines, rather than on the yellow fill, to understand how it works. The **`Polyline`** is a very powerful **`Shape`** object, but it requires more experimenting to get the most out of it. This is left to you as an exercise.

## Further studies: Path and geometries

For very complex shapes, including curves, Xamarin.Forms provides the **`Path`** class and several **`Geometry`** objects. Their usage can be useful in game development and in applications that need to represent mathematical calculations, but they are also complex to implement. The **`Path`** class inherits the properties described in *Table 10.1* and defines a shape via its **`Data`** property. The following code example is taken from the official documentation (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/shapes/path**) and is an appropriate one to understand the **`Path`** class:

```
<Path Data="M 10,100 L 100,100 100,50Z" Stroke="Black"
Aspect="Uniform"
    HorizontalOptions="Start" />
```

The **`Data`** property includes some commands, better described as follows:

- **`M`**, known as the *move* command, represents the coordinates of the absolute starting point for the path.
- **`L`**, known as the *line* command, draws a line that connects the **`M`** point to the specified end point.
- **`Z`**, known as the *close* command, connects the current point to the **`M`** point.

The preceding code draws a triangle, as shown in *Figure 10.8*:

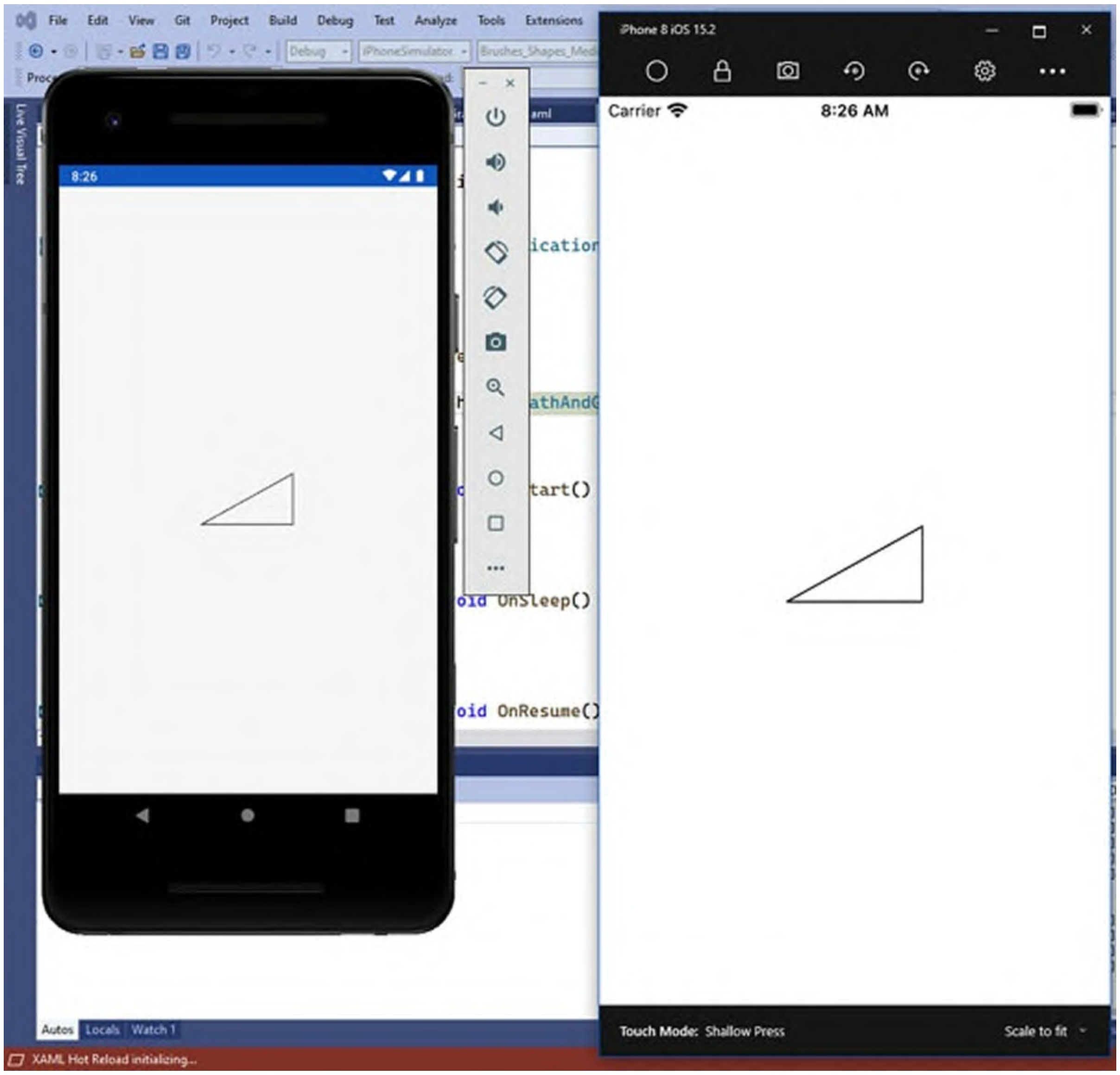


*Figure 10.8: Drawing custom shapes with the Path*

The **`Path`** class can draw much more complex shapes, and its **`Data`** property can also be assigned with **`Geometry`** objects. These are special objects that are optimized to draw 2D shapes and allow for drawing both simple and complex geometries. Among the simple geometries, there are the **`EllipseGeometry`**, **`LineGeometry`**, and **`RectangleGeometry`**, whose names are self-explanatory. For example, the following code draws an ellipse, a line, and a rectangle:

```
<Path Fill="Yellow" Stroke="Red">
  <Path.Data>
    <EllipseGeometry Center="50,50" RadiusX="50" RadiusY="50" />
  </Path.Data>
</Path>
<Path Stroke="Black">
```

```
  <Path.Data>
   <LineGeometry StartPoint="10,20" EndPoint="100,130" />
  </Path.Data>
</Path>
<Path Fill="Yellow" Stroke="Red">
  <Path.Data>
   <RectangleGeometry Rect="10,10,150,100" />
  </Path.Data>
</Path>
```

The following list summarizes the key properties:

- For the **`EllipseGeometry`** object, you control the center of the ellipse with the **`Center`** property, a coordinate of type **`Point`**, and the radius with the **`RadiusX`** and **`RadiusY`** properties of type double.
- For the **`LineGeometry`** object, you define the line boundaries via the **`StartPoint`** and **`EndPoint`** properties, but the position of the line is determined by the **`Path`** position.
- For the **`RectangleGeometry`** object, the **`Rect`** property defines the boundaries of the rectangle.

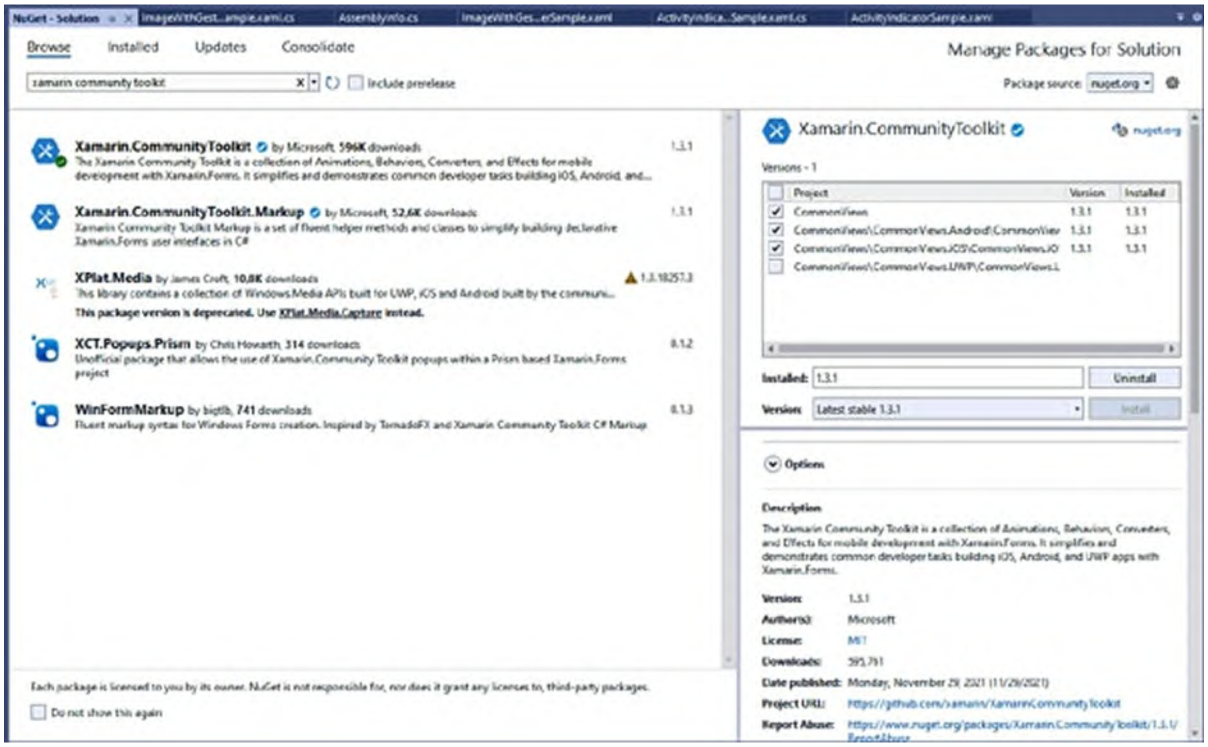
There is another **`Geometry`** object called **`PathGeometry`**, which makes it possible to create extremely complex shapes, including curves and arcs. This is not covered in this chapter due to its usage in very specific scenarios, but the official documentation provides a good overview with examples and can be found at **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/shapes/geometries**.

## Working with multimedia

Playing videos and audio is another common requirement, not only with media-oriented applications but also inside user interfaces that want to engage users with appealing content. The Xamarin.Forms code base does not include a cross-platform media player, but Microsoft recently developed one called **`MediaElement`** and included it in the Xamarin Community Toolkit (**https://docs.microsoft.com/en-us/xamarin/community-toolkit**), an open source library that contains many reusable views, converters, behaviors, and so on. In this book, the Community Toolkit will only be mentioned a few times, but an exception is made for the **`MediaElement`** because the demand to have this view has always been very high, and because including media is really a common requirement in mobile apps. Free and paid third-party components also exist, but this book focuses only on what is available in the family of Microsoft libraries.

## Installing the Xamarin Community Toolkit

The first thing you need to do is install the Xamarin Community Toolkit as a NuGet package into your solution. To accomplish this, in **`Solution Explorer`**, right-click on the solution name and then select **`Manage NuGet Packages for Solution`**. This will open the NuGet Package Manager, a tool that allows for installing additional libraries and that automatically resolves dependencies for each of the libraries that you want to install. In the search box of the **`Browse`** tab, type **`Xamarin Community Toolkit`** (see *Figure 10.9*).


NuGet - Solution
ImageWithGest...ample.xaml.cs
AssemblyInfo.cs
ImageWithGes...erSample.xaml
ActivityIndica...Sample.xaml.cs
ActivityIndicatorSample.xaml
Browse
Installed
Updates
Consolidate
Manage Packages for Solution
Include prerelease
Package source: nuget.org
Options
Uninstall
Install
Do not show this again

xamarin community toolkit

| Package | Version |
|---|---|
| **Xamarin.CommunityToolkit** by Microsoft, **596K** downloads<br>The Xamarin Community Toolkit is a collection of Animations, Behaviors, Converters, and Effects for mobile development with Xamarin.Forms. It simplifies and demonstrates common developer tasks building iOS, Android, and... | 1.3.1 |
| **Xamarin.CommunityToolkit.Markup** by Microsoft, **52.6K** downloads<br>Xamarin Community Toolkit Markup is a set of fluent helper methods and classes to simplify building declarative Xamarin.Forms user interfaces in C# | 1.3.1 |
| **XPlat.Media** by James Croft, **10.8K** downloads<br>This library contains a collection of Windows.Media APIs built for UWP, iOS and Android built by the communi...<br>**This package version is deprecated. Use XPlat.Media.Capture instead.** | 1.3.18257.3 |
| **XCT.Popups.Prism** by Chris Hovarth, **314** downloads<br>Unofficial package that allows the use of Xamarin.Community Toolkit popups within a Prism based Xamarin.Forms project | 0.1.2 |
| **WinFormMarkup** by bigtlb, **741** downloads<br>Fluent markup syntax for Windows Forms creation. Inspired by TornadoFX and Xamarin Community Toolkit C# Markup | 0.1.3 |

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Xamarin.CommunityToolkit — nuget.org

Versions - 1

| | Project | Version | Installed |
|---|---|---|---|
| ✓ | CommonViews | 1.3.1 | 1.3.1 |
| ✓ | CommonViews\CommonViews.Android\CommonViev | 1.3.1 | 1.3.1 |
| ✓ | CommonViews\CommonViews.iOS\CommonViews.iO | 1.3.1 | 1.3.1 |
| | CommonViews\CommonViews.UWP\CommonViews.L | | |

Installed: 1.3.1

Version: Latest stable 1.3.1

Description

The Xamarin Community Toolkit is a collection of Animations, Behaviors, Converters, and Effects for mobile development with Xamarin.Forms. It simplifies and demonstrates common developer tasks building iOS, Android, and UWP apps with Xamarin.Forms.

| | |
|---|---|
| Version: | 1.3.1 |
| Author(s): | Microsoft |
| License: | MIT |
| Downloads: | 595,791 |
| Date published: | Monday, November 29, 2021 (11/29/2021) |
| Project URL: | https://github.com/xamarin/XamarinCommunityToolkit |
| Report Abuse: | https://www.nuget.org/packages/Xamarin.CommunityToolkit/1.3.1/ReportAbuse |

***Figure 10.9:*** *Installing the Xamarin Community Toolkit*

On the left side, select the **`Xamarin.CommunityToolkit`** package and on the right side of the package manager, make sure that the shared Android and iOS projects are selected as targets. By default, NuGet proposes the latest version of a package, and, at the time of writing this book, the latest version available of the Toolkit is 1.3.1. When ready, click on **`Install`** and accept the necessary license agreements when prompted. The installation will take just a few seconds.

## Implementing the MediaElement

In the page where you want to use the **`MediaElement`**, you first need to add an XML namespace definition as follows:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:toolkit="clr-
  namespace:Xamarin.CommunityToolkit.UI.Views;assembly=Xamarin.Co
  x:Class="CommonViews.MediaElementExample">
```

The **`MediaElement`** supports streaming media files from remote URIs, from the local device library, from embedded resources, and from local folders. The following example shows how to play a remote, free, and public video about Xamarin from Microsoft:

```
<toolkit:MediaElement ShowsPlaybackControls="True"
x:Name="Media1"
  Source="https://sec.ch9.ms/ch9/5d93/a1eab4bf-3288-4faf-81c4-
  294402a85d93/XamarinShow_mid.mp4"/>
```

The video or audio file you want to play is assigned to the **`Source`** property. The **`MediaElement`** invokes the platform-specific media players, so you can assign the **`ShowPlaybackControls`** property with true and leverage the play, pause, stop, and transport controls from the native players. If you set this property to false, you will have the chance to design your own buttons and invoke the **`Play`**, **`Pause`**, and **`Stop`** methods that the **`MediaElement`** exposes. By default, the media file starts playing as soon as it is loaded (or as soon as buffering starts, in case of remote files). You can assign the **`AutoPlay`** property with false if you want to avoid this behavior. *Figure 10.10* shows the **`MediaElement`** in action.

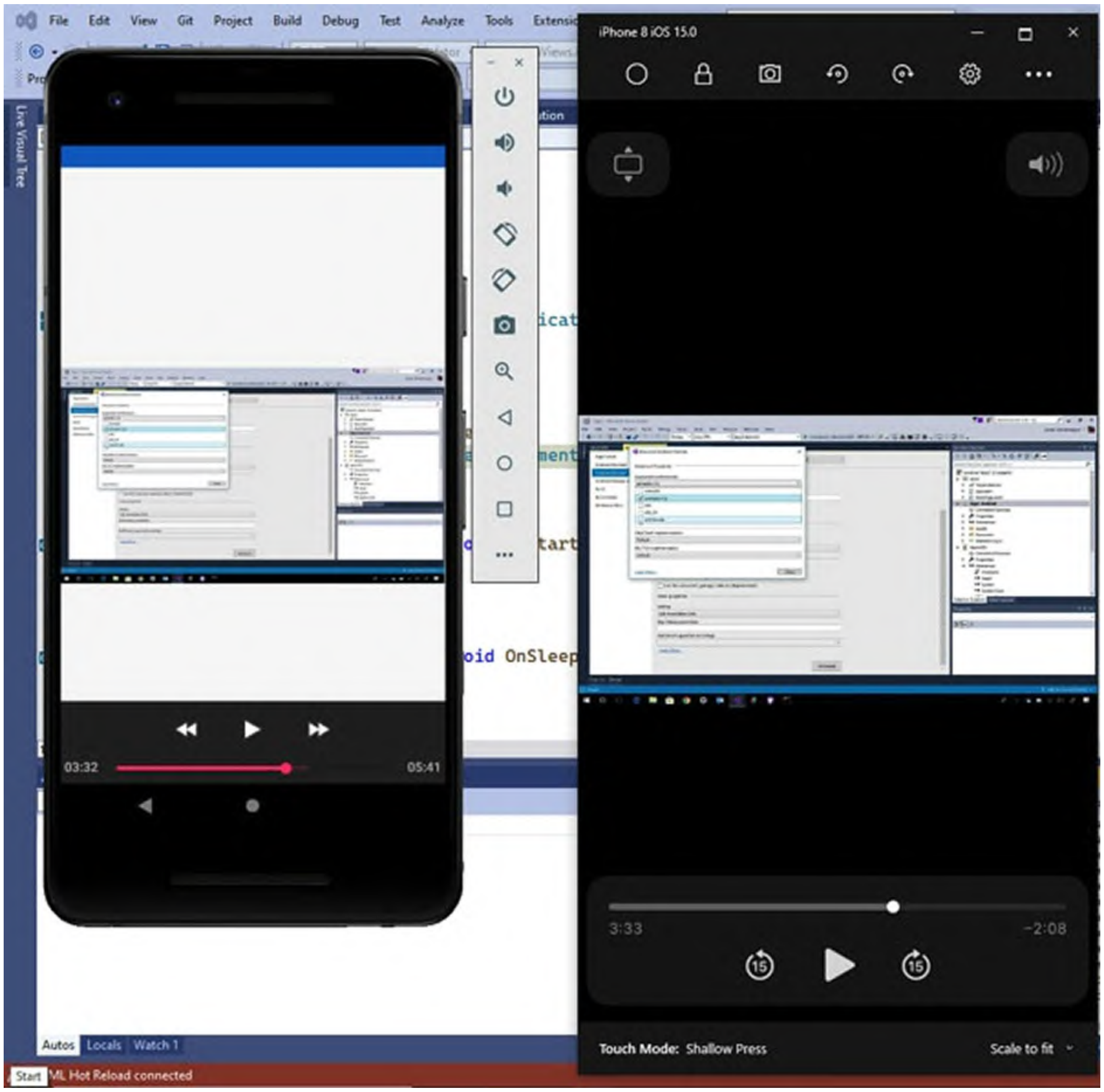


***Figure 10.10:*** *Playing media files*

If screen rotation is enabled on the device, the video will also rotate for wider view. Because the **`MediaElement`** relies on the native players, supported media formats also depend on the target system.

## Controlling the media file

The **`MediaElement`** exposes several members that you can use to manage and control the media file. These are summarized in *Table 10.2*:

| Member | Description |
|---|---|

| | |
|---|---|
| **`Volume`** | Of type `double`, this property controls the media volume with a value between 0 and 1. |
| **`Position`** | Of type `double`, this property returns the current position over the duration. It is updated every 200ms. It can be changed in C# to move the video to another position. |
| **`CurrentState`** | Of type `MediaElementState`, this property returns the state of the player. Self-explanatory values are closed, opening, buffering, playing, paused, and stopped. |
| **`MediaOpened`** | An event that is raised when the media file was successfully opened. |
| **`MediaEnded`** | An event that is raised when the media reproduction ended. |
| **`MediaFailed`** | An event that is raised if opening the media file fails. |
| **`Play`** | A method that allows for manually playing a media file. |
| **`Pause`** | A method that allows for pausing the media reproduction. |
| **`Stop`** | A method that stops the media reproduction. |
| **`ShowPlaybackControls`** | Of type `bool`, this property shows or hide the system controls. |
| **`AutoPlay`** | Of type `bool`, this property sets automatic start of playing. |
| **`Duration`** | Of type `TimeSpan`, this property returns the duration of the media content. |
| **`Aspect`** | Of type `Aspect`, it allows for sizing and stretching the video with the same values (`AspectFit`, `AspectFill`, `Fill`) and behaviors described for the Image view. |

***Table 10.2:*** *Most relevant members of the MediaElement*

For example, suppose you have set the **`AutoPlay`** property as false. When you assign the Source property of the **`MediaElement`**, the video starts loading but you need to play it manually. You could write the following code:

```
private void Media1_MediaOpened(object sender, EventArgs e)
{
 Media1.Play();
}
private async void Media1_MediaFailed(object sender, EventArgs
e)
{
 await DisplayAlert(“Error", “There was a problem while opening
 your media",
  “OK");
}
```

The preceding code shows how to leverage the **`MediaOpened`** and

**MediaFailed** events. In the first case, the **Play** method is invoked to start playing the media. In the second case, a warning message is shown to the user.

## Playing local files

The source of the **MediaElement** can be an embedded video, a local file, and a URI. For the last, you just saw an example. In the case of a file embedded in the app resources, you will use the following syntax:

```
<MediaElement Source="ms-appx:///YourVideo.mp4" />
```

You basically add the **ms-appx:///** prefix to the filename. With a similar syntax, you can also play media files stored in the app's local or temporary folder. You can include files in the local app folder by copying them into the **Resources** folder of the Xamarin.iOS project and into the **Assets** folder of the Xamarin.Android project. For example:

```
<MediaElement Source="ms-appdata:///local/YourVideo.mp4" />
```

Plays a video in the app folder, whereas <MediaElement Source="ms-appdata:///temp/YourVideo.mp4" /> plays a video in the temporary folder. They have in common the **ms-appdata:///** prefix, followed by the folder name. Finally, it is possible to play media stored in the device's library. To accomplish this, you first need a user interface that picks up the content from the library. The Xamarin Community Toolkit implements shared access to the device specific folders, so you could write the following C# code, for example, inside the **Clicked** event handler of a **Button**:

```
string mediaName = await DependencyService.Get<IVideoPicker>
().GetVideoFileAsync();
if (!string.IsNullOrWhiteSpace(mediaName))
{
 MediaElement1.Source = new FileMediaSource
 {
  File = mediaName
 };
}
```

The **DependencyService** class will be detailed in *Chapter 13: Working With Native API*, but for now, what you need to know is that it allows for invoking platform-specific code from the shared project. Because the system API that allows for picking up contents from the device are certainly different on iOS and Android, there are separate implementations of the **GetVideoFileAsync** method, whose purpose is showing a file picker and allowing users to select a media file. The resulting name, if not null, is assigned to the `Source` property of the `MediaElement` via an instance of the `FileMediaSource` class. Like for the `Image`, you can also programmatically set the source of the `MediaElement` via the `FromFile` and `FromUri` methods, exposed by the `MediaSource` class.

## Conclusion

Making applications appealing is crucial for a mobile app developer. Xamarin.Forms provide brushes and shapes that not only allow for creating gradients and geometries but that can be also used to implement interesting user experiences. Think of the example of an avatar, with an image inside an ellipse. Another crucial point in mobile apps today is providing the ability to play audio and videos. In Xamarin.Forms, support for this has been added recently with the `MediaElement` view, offered through the Community Toolkit, an open-source library backed by Microsoft and that you might want to bookmark for further studies. Creating beautiful applications is important, but it is not enough. In fact, you need to learn how an application works from start to end, which is the topic of the next chapter.

## Key terms

- **Gradient (for app development)**: A scalar background made of several colors.
- **Geometry**: An object that represents both simple and composite geometrical shapes.
- **Dependency service**: A class that allows for accessing native code from the shared project.

## Suggested readings

- Microsoft official documentation about brushes (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/brushes/**)
- Microsoft official documentation about shapes (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/shapes/**)

- Microsoft official documentation about the Xamarin Community Toolkit (**https://docs.microsoft.com/en-us/xamarin/community-toolkit/**)

# CHAPTER 11
# Managing the Application Lifecycle

## Introduction

Simple things an application does, or simple gestures of the user done on an app, have a reaction in the so-called **application lifecycle**. This represents moments in time in which the app is working or when it is being closed. When the user opens an app, the startup is the first moment of the application lifecycle. When the user closes an app, it is the last moment of the application lifecycle. When the app is sent to the background, for example, when it is not shut down and the user just switches to another app, this is yet another moment in the application lifecycle. Understanding and properly managing the application lifecycle is crucial in mobile app development because you need to know how to keep the information of key importance for the user. For example, suppose your app has a data form where the user is entering many details. If the app is sent to the background, you need to ensure that the information is shown again once the app is reopened. This chapter explains the application lifecycle events from a Xamarin perspective, and it also enhances your knowledge of the Model-View-ViewModel pattern with techniques that are still considered part of the application lifecycle, such as working with broadcast messages.

## Structure

In this chapter, we will cover the following topics:

- The Application class
- Events of the Application lifecycle
- Sending messages through the app

## Objectives

After completing this chapter, you will be able to fully manage the application lifecycle, understanding where and when to persist user settings and preferences before the app shuts down.

## Creating a sample project

This chapter comes with a companion **Xamarin.Forms** solution that you can open with Visual Studio to better follow the examples. If you wish to create a project on your own from scratch, you can follow these steps:

1. Create a new **Xamarin.Forms** solution called **ApplicationLifecycle** for consistency with the sample solution.
2. Do not edit or remove the auto generated **MainPage.xaml** file; it will be used later.
3. For each layout discussed in the book, add a new item of type **Content Page (XAML)**. To accomplish this, right-click on the shared project name and then click on **Add New Item** in the **Context** menu.
4. In the **Add New Item** dialog, click on the **Xamarin.Forms** node on the left and then select the **Content Page (XAML)** item template.
5. Assign to the new XAML file a name that matches the discussed view, for example, **ApplicationEvents.xaml**, and click on **Add**.

For each page you add to the project, add an empty **StackLayout** to the **ContentPage** and assign its **VerticalOptions** property with **CenterAndExpand**. Unless specified, this will be the layout of choice for the code examples in the next pages.

## The Application class

The **Application** class represents the running instance of the application from a coding perspective. It is responsible for the application lifecycle and responds to events such as app startup and page opening. Every **Xamarin.Forms** solution contains an **App** class that derives from **Application**, and you actually work against **App** rather than **Application**. The definition of this class is inside the **App.xaml.cs** file; it also defines resources that are available across the application.

**With regard to resources, in *Chapter 9: Resources and Data Binding*, you learned how you place all the resources you want to share across**

**the app in the `App.xaml` file. Behind the scenes, resources are handled by the `App` class and its `Resources` property. So, XAML simply provides an easier interface to define and consume resources.**

Because the **`App`** class is singleton, which means that only one instance can exist, it exposes a property called **`Current`**, of the **`App`** type, which you use to interact with the class itself. The following line shows how to use **`Current`** to retrieve the device's graphical theme:

```
var theme = App.Current.UserAppTheme;
```

Many types and members of the **`App`** class are for the platform's internal usage, but a few are extremely important for you. For example, the **`MainPage`** property, of the **`Page`** type, represents the root page for the application. You already used this property several times in this book, but it is important to remember that it is part of the application lifecycle. When you create a project, the assignment looks like the following:

```
public App()
{
  InitializeComponent();
  MainPage = new MainPage();
}
```

Not only can you change the root page here, but you can also reassign the main page later, simulating an application restart. In this case, you can change the root page as follows:

```
App.Current.MainPage = new SecondaryPage();
```

The **`App`** class is also useful to make changes that impact the application and to store fields and properties that you want to make available everywhere. The following sections discuss these scenarios.

## Working with themes

Android and iOS have graphical themes, and they share the existence of a dark and a light theme. You can easily retrieve which theme is currently active on the user's device, and you can even set a different theme for the app through the **`App`** class. The following code demonstrates how to retrieve the theme:

```
var currentTheme = App.Current.UserAppTheme;
switch(currentTheme)
{
  case OSAppTheme.Light:
   break;
  case OSAppTheme.Dark:
   break;
  default:
   break;
 }
```

The **`UserAppTheme`** property specifies the theme and is of the **`OSAppTheme`** type, an enumeration that exposes three values: **`Light`**, **`Dark`**, and **`Unspecified`**. The following code demonstrates how you can force the app to use a specific theme instead:

```
 App.Current.UserAppTheme = OSAppTheme.Light;
```

## Defining global variables

In some situations, you might need to store values or references into variables that you want to make available across the app. To accomplish this, you can define static fields or properties in the **`App`** class, like in the following example:

```
 public partial class App : Application
 {
  internal static string SharedProperty { get; set; }
  …
 }
```

In this case, the **`SharedProperty`** property is marked as internal so that it is only accessible from within the shared project. In addition, it is marked as static because there can only be one instance of the **`App`** class. This is a particular situation because you access these members without invoking the **`Current`** property; so, you would use it as follows:

```
 App.SharedProperty = "Some text";
```

Variables that you can access from anywhere in the app are known as global variables.

**Tip: Global variables are useful and convenient in some situations, but keep in mind that they take up allocated space in memory even when you are not using them. For this reason, you should avoid global variables when possible and prefer local variables, which are destroyed when no longer used.**

## Events of the Application lifecycle

The **`Application`** class exposes .NET events that represent the most important moments in the lifecycle of an application. This section discusses events that are strictly related to the working time of the app and events that are raised when some actions are taken by the user.

## Understanding and using Application events

The **`Application`** class exposes the following three events:

- **`OnStart`**, which is fired when the app has started (and it was previously shut down).
- **`OnSleep`**, which is fired when the app is sent to the background or when it is shut down.
- **`OnResume`**, which is fired when the app is brought back to the foreground.

You can handle these events to execute actions in those specific moments of the app lifecycle; for example, loading user settings when **`OnStart`** is fired or persisting user settings when **`OnSleep`** is fired. The event handlers are not added by Visual Studio, so you need to add them manually to the **`App.xaml.cs`** file, as follows:

```
protected override void OnStart()
{
}
protected override void OnSleep()
{
}
protected override void OnResume()
{
}
```

You will now learn a real usage of these events, which is common to many applications.

### A real-world example: Storing and retrieving data

A common usage of the application events is retrieving and storing the last time the user has accessed the application. Reasons for doing this can be infinite; for example, you might want to ask the user to log in again after a certain amount of time has passed with the application being in the background. To accomplish this, you will use the **`Xamarin.Essentials`** library, which is automatically added by Visual Studio to a new solution and will be covered in further detail in *Chapter 13: Working with Native API*. This library exposes a class called **Preferences**, which allows for saving primitive information to the app's local storage. You can use the **Get** method of the class to retrieve information and the **Set** method to save it. The following code demonstrates this:

```
private DateTime _lastActivityTime;
protected override void OnStart()
{
  _lastActivityTime = Preferences.Get(“LastActivityTime”,
  DateTime. MinValue);
}
protected override void OnSleep()
{
  Preferences.Set(“LastActivityTime”,
  DateTime.Now.ToUniversalTime());
}
protected override void OnResume()
{
  _lastActivityTime = Preferences.Get(“LastActivityTime”,
  DateTime. MinValue);
}
```

When the app starts and **OnStart** is fired, the code searches for a variable called **LastActivityTime** in the app's local storage. If found, it returns the stored value. If not found, for example, because it is the first time the app is running, a default value is returned. When the app is sent to the background or is shut down, **OnSleep** is fired, and the current date and time is saved to the app's local storage. When the app is resumed from the background, **OnResume** is raised, and the code searches for the same date and time value and returns a default value if nothing is found. Another possible usage of the application events is saving all the data the user has entered into the app in a database before it is sent to the background or shut down so that when the app is resumed, nothing is lost. This is not just a best practice; it is how an app should work to offer the proper user experience.

## A real-world scenario: Restoring data forms

There are applications that provide data forms and that show pending data at the next startup if the data was not saved before the app was closed or suspended. This is not a mandatory behavior, and it depends on the requirements and design for the app. If you must address this scenario, you have plenty of ways to do it, depending on how complex the data structure is. For example, if you have one instance of an object that you need to persist, you could save it to a local JSON file that is serialized back to a .NET object when the application starts up again. If you are also working with a local SQLite database, you could create a specific table for temporary data and save it when the application is suspended or closed. Then, at startup, you can load data back from the temporary table, which you will clean once data is finally saved to the proper table. Usually, the approach you use is shared with the development team, and it depends on your requirements. In many applications, it is enough to warn the user about the fact that pending data will be lost if not saved before the app is closed or suspended.

## Responding to page events

Sometimes, you might want to know when a page is opened or closed. In such cases, the **`Application`** class exposes the events described in *Table 11.1*:

| Member | Description |
|---|---|
| **`PageAppearing`** | Raised when a page is being rendered onscreen |
| **`PageDisappearing`** | Raised when a page is being removed from screen |
| **`ModalPushing`** | Raised when a modal page is being added to the navigation stack |
| **`ModalPushed`** | Raised when a modal page has been rendered onscreen |
| **`ModalPopping`** | Raised when some code has invoked `PopModalAsync` over a modal page |
| **`ModalPopped`** | Raised when a modal page is removed from the navigation stack |

***Table 11.1:*** *Events related to page navigation*

When working with these events, you need to explicitly subscribe to them, like in the following example:

```
App.Current.PageAppearing += Current_PageAppearing;
```

For **`PageAppearing`** and **`PageDisappearing`**, the event handler gets the instance of the page:

```
private void Current_PageAppearing(object sender, Page e)
{
  if(e is MainPage)
  {
  }
}
```

In this code snippet, the code checks whether the page is the **`MainPage`** object. For the other events, the event handlers' signatures work with objects of the **`ModalPushingEventArgs`**, **`ModalPushedEventArgs`**, **`ModalPoppingEventArgs`**, and **`ModalPoppedEventArgs`** types, respectively. All these classes expose the **`Modal`** property, of the **`Page`** type, which refers to the instance of the modal page that raised the event. The following snippet provides an example:

```
private void Current_ModalPopping(object sender,
ModalPoppingEventArgs e)
{
  if(e.Modal is SecondaryPage)
  {
  }
}
```

**Tip: Remember the `OnAppearing` and `OnDisappearing` events raised by individual pages when you need to work at the page lifecycle level.**

## Sending messages through the app

It happens very often that objects in different parts of your apps need to communicate with one another. For example, a method in a **`viewmodel`** needs to communicate to a view that an operation was completed so that the view can take the appropriate actions. This happens via broadcast messages, which, in **`Xamarin.Forms`**, you manage via the static **`MessagingCenter`** class. This class works with a publisher/subscriber model, meaning that an object, the publisher, sends a broadcast message without knowing which other object is going to receive the message, and other objects, the subscribers, listen to a specific message to take the necessary actions when intercepted. The **`MessagingCenter`** class exposes three main methods: **`Send`**, **`Subscribe`**, and **`Unsubscribe`**. In its simplest form, a message is sent with this syntax:

```
MessagingCenter.Send(this, “MESSAGE”);
```

Send takes two parameters: the instance of the object that is sending the message, and a string containing the message. The message string is completely your choice. Typically, the instance of the object that sends the message is the same that is invoking **`Send`**. A subscriber registers for a message as follows:

```
MessagingCenter.Subscribe<MainPage>
```

```
  (this, “MESSAGE”, (sender) =>
  {
   // Do something here
  });
```

The **`Subscribe`** message takes a **`type`** parameter that represents the object that is sending the message; in the example, it is **`MainPage`**. The first method parameter is then the instance of the object that is subscribing to the message, the message that the object is subscribing to, and an action that receives the instance of the sender as an argument. Remember to unsubscribe from messages when no longer needed, with the following code:

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

**`Unsubscribe`** needs you to specify the type of the sender, and then it takes the instance of the subscriber and the message as parameters.

**Tip: Unsubscribing from messages is extremely important. If you dispose of an object instance without unsubscribing from the messages it was listening to, and then you create a new instance of the same object, there might be runtime conflicts, and messages might not be subscribed again as expected.**

There are many scenarios in which you can use broadcast messages, but Model-View-ViewModel is certainly the most common. Now, you will learn how to improve MVVM architectures.

## Broadcast messages with MVVM

Suppose you have a method in a viewmodel that completes an action, and that you want to show a message to the user by displaying an alert or a new page. Based on the MVVM principles, a viewmodel cannot open a view or an alert and a data-bound view should only implement code that is related to the user interface, without direct interaction with the viewmodel. In such situations, when a viewmodel completes an action, it can send a message. Views can subscribe to that message and take the appropriate actions such as displaying an alert or a new page, when it is sent. For better understanding and practical work, in Visual Studio, reopen the **`LocalDataAccess`** sample solution discussed in *Chapter 9: Resources and Data Binding*. When ready, open the **`ContactViewModel.cs`** file and locate the following code:

```
SaveAllCommand = new Command(() =>
ContactsDataBase.SaveAll(Contacts));
```

As you might remember, this command invokes the **SaveAll** method that stores all pending changes to a list of contacts in the SQLite database. Now, imagine that you want to notify users of the completion with an alert. You can extend the code given earlier as follows:

```
SaveAllCommand = new Command(() =>
{
  ContactsDataBase.SaveAll(Contacts);
  MessagingCenter.Send(this, “ContactsSaved”);
});
```

A message is sent when the save operation is completed. Now, open the **MainPage.xaml.cs** file and change the constructor as follows:

```
public MainPage()
{
  InitializeComponent();
  MessagingCenter.Subscribe<ContactViewModel>(this,
  “ContactsSaved”,
    async (sender)=> {
      await DisplayAlert(“Information”, “Contacts saved!”, “OK”);
    });
  ViewModel = new ContactViewModel();
  BindingContext = ViewModel;
}
```

The **Subscribe** method is invoked to register for the **ContactsSaved** message sent by the **ContactViewModel** class, and the action that is taken is displaying an alert. With this approach, your code is fully adhering to the MVVM principles and is implementing a common feature, that is, displaying additional user interface when an action in the viewmodel is completed. *Figure 11.1* shows how the alert appears, thanks to the message exchange:

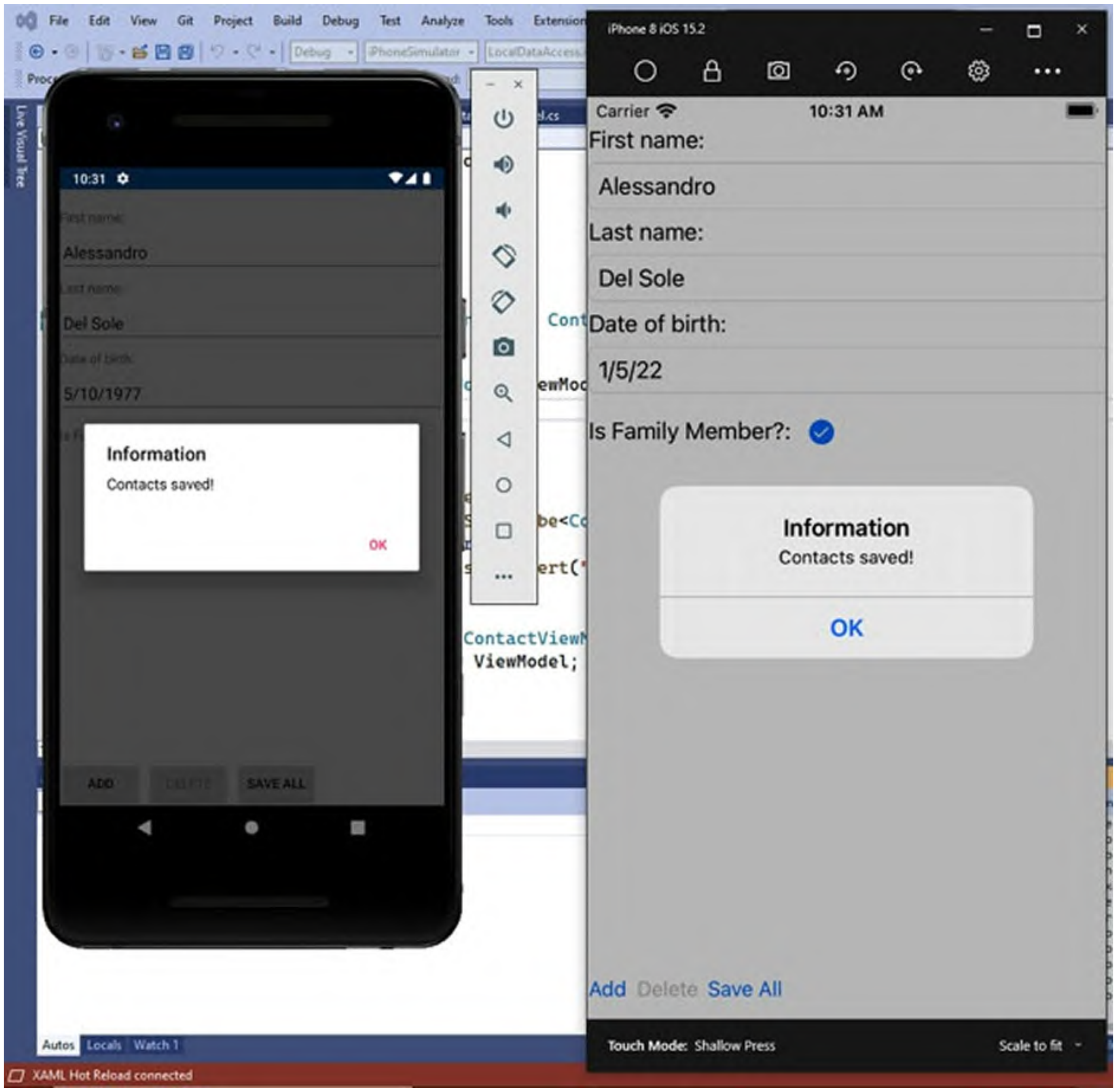


***Figure 11.1:*** *Displaying an alert via message exchange*

You are working with only one page in the sample solution, so invoking **`Unsubscribe`** is not necessary as destroying the page instance means closing the app. However, if you were on a different page that is closing, invoking **`Unsubscribe`** is important.

## MessagingCenter tips and tricks

There are some best practices and suggestions that you should know about broadcast messages. The first suggestion is to invoke **`Subscribe`** before creating an instance of the object that sends the message. If you look back at the constructor of the **`MainPage`** object, you can see this. This is not mandatory, but if the constructor of the sender object already sends messages, you would lose them. A second suggestion is about message strings. If you want to avoid typos and take advantage of Visual Studio’s code editor feature, you can define message strings as constants in a separate class, as follows:

```
internal static class MessageConstants
{
  public const string ContactsSaved = “ContactsSaved”;
}
```

You can then refer to the message via the constant, which enables IntelliSense in Visual Studio, speeding up adding the string and avoiding typos. The following line is an example:

```
MessagingCenter.Send(this, MessageConstants.ContactsSaved);
```

The last tip is about the actions that are executed by the subscriber. In the previous code examples, the action is encapsulated inside a lambda expression. However, if you have more complex code to execute, you can rewrite the subscriber as follows:

```
MessagingCenter.Subscribe<ContactViewModel>(this,
“ContactsSaved”,
    ShowCompletionMessageAsync);
```

You are basically passing the name of a method that must be invoked when the message is received. Consider the following example:

```
private async void ShowCompletionMessageAsync(ContactViewModel
contactViewModel)
{
  await DisplayAlert(“Information”, “Contacts saved!”, “OK”);
}
```

This approach makes it clearer how the instance of the sender object is also received by the subscriber, and it provides a better view of the code, which is recommended when you have more lines to handle.

## Conclusion

Managing the application lifecycle is very important when developing mobile apps. Not only does this give you full control over the application, but it also allows you to implement the proper user experience according to the moment of the life of an app. In this chapter, the application lifecycle events have been useful to extend your knowledge about working with local data.

However, most mobile apps work with remote data and services. So, in the next chapter, you will work with Web API services and enhance your understanding of C#, .NET, and Xamarin.Forms.

## Key terms

- **Application lifecycle**: Represents the working time of the app, from when it starts to when it is shut down or sent to the background.
- **Global variables**: Variables that are available to all the objects in the app.
- **Broadcast message**: A string message that is exchanged between objects in the app.

## Suggested readings

Microsoft official documentation about the application lifecycle (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/app-lifecycle**).

# CHAPTER 12
# Working with Web API

## Introduction

In the previous chapters, you have seen how to work with local data and local SQLite databases to read and write information. However, in the real world, most data-driven applications also work with remote data sources. In a typical architecture, a web service is invoked to read from and write data to a remote database, and both the web service and the database can be hosted either on a server or on a cloud service. In a perfect world, applications and web services should be able to communicate with each other regardless of the development technology and programming languages used to create them, so the data they exchange should be provided in a universal format. In this chapter, you will learn how to create and consume web services with the most modern Microsoft technologies, exchanging data with a mobile application using standard data exchange formats.

## Structure

In this chapter, we will cover the following topics:

- Chapter prerequisites
- Understanding web services and Web API
- JSON: A standard data exchange formats
- Creating Web API in Visual Studio
- Publishing Web API services
- Consuming Web API with Xamarin.Forms

## Objectives

After completing this chapter, you will extend your knowledge with important concepts related to data-driven development, and you will be able to implement data exchange with remote data sources in your Xamarin.Forms projects.

**Unlike the previous chapters, where the companion solution is made of one project, for this chapter, the solution is made up of multiple projects. For this reason, the steps for creating projects will be described in the dedicated sections. You can open the companion `BookService` sample solution if you wish to follow the explanations without manual steps.**

## Chapter prerequisites

In the first part of this chapter, you will learn how to create and publish a Web API service to a remote server. It is not possible to predict the environment you are using, so, in order to offer the same possibilities to all readers, this chapter will use Microsoft Azure as the remote destination, which is also useful to understand how easy it is to publish and manage a Web API service on the cloud. Instead, if you prefer to publish your services to an on-premises server, you will likely need to contact the system administrator of the company, who will help you set up and configure the necessary infrastructure. In addition, you will use a tool called Postman to test the API calls you will implement, because it is a very popular tool used by many companies.

## Getting a Free Azure subscription

Azure is the cloud platform from Microsoft that provides a wide range of services. Azure is not free, but it is possible to obtain a free trial at **https://azure.microsoft.com/en-us/free** if you do not already have one. A free trial subscription is enough to complete the steps described later. You will just need to follow the steps described on the website, signing in with a Microsoft account. You might be asked to enter a valid credit card, which is only used to verify that you are a real entity (but everything is explained clearly). By visiting the website, you will also be able to discover the full offer of cloud services, though here you will just use the web application service.

## Downloading Postman

Postman (**https://www.postman.com**) is a free tool that developers use to build and test API calls. It is a very powerful tool, and it supports many data exchange formats, environments, and security standards. It is probably the most popular tool that developers use to work with web API, and even the Visual Studio debugger can detect if an API call is made by Postman and will allow you to use the usual debugging tools accordingly. For now, download and install Postman; the installer is based on a one-click approach, so it is very easy. Later in this chapter, you will use it to test your work.

## Understanding web services and Web API

A web service is a software that allows for communication between a source and a client application, and vice versa, over a network. A web service is made available to clients by publishing it to a remote server. By remote server, we mean both an on-premises server and a cloud service like Microsoft Azure. The communication is made possible by functions that a web service exposes to the public and that clients can invoke as they would do with any .NET method. The result of such invocations is the data that the client and service exchange. Web services can be written with different technologies and should be architected in a way that they can be consumed by different types of client applications (for example, desktop apps, mobile apps, and so on). From a Microsoft perspective, .NET has always made it possible to create and publish web services with C# (and the other .NET languages) through different technological flavors: ASP.NET and ASP.NET Core, **Windows Communication Foundation** (**WCF**) and, more recently, ASP.NET Web API.

**Web API can be created with any development platform that supports REST. However, in this chapter, any reference to Web API means ASP.NET Core Web API and, therefore, to services built with the Microsoft stack.**

WCF, in particular, has represented an important milestone in the .NET evolution, because it is a technology that allows to leverage all the power of .NET on the server side, it supports a large number of communication protocols and the connection between clients and services is simplified by specific tools in Visual Studio. On the other side, WCF implements services and communication in a way that derives from the **Simple Object Access Protocol** (**SOAP**) format, but that is only supported by applications built with .NET. In mobile development, this is not the recommended approach, and you should prefer web services that can be consumed by clients written with different technologies. To accomplish this, you use Web API services. Generally speaking, a Web API is a web service that you access through the HTTP and HTTPS protocols; functions they expose are identified by a static URL called **endpoint**. For example, if a Web API called **`CustomerService`** is hosted on a server called **`Server01`**, the address for the Web API will be as follows:

```
https://server01/CustomerService
```

If the service exposes a function used to retrieve a list of customers, called **`GetCustomers`**, the endpoint (that is, the address of the function) will be as follows:

```
https://server01/CustomerService/GetCustomers
```

Functions support parameters, but this will be discussed later in the chapter. Applications send requests to a web API service through the so-called **REpresentational State Transfer** (**REST**) approach, and .NET provides objects that allow to send and manage requests in an object-oriented way. REST is an architectural style for designing web software based on constraints. It is not necessary to delve into the details of REST, but it is important to underline that every request is identified by a specific HTTP verb. The most common HTTP verbs represent operations against data and are described in *Table 12.1*:

| HTTP Verb | Description |
|---|---|
| **`GET`** | Reads and returns data from the data source. |
| **`POST`** | Writes new data to the remote source. |
| **`PUT`** | Modifies existing data in the remote source. |
| **`PATCH`** | Partially modifies existing data in the remote source. |
| **`DELETE`** | Deletes data from the remote source. |

***Table 12.1:*** *Common HTTP verbs*

Actually, HTTP verbs do not execute operations directly. The Web API that receives a request detects the HTTP verb specified in the request and reacts accordingly. This means you could use a **`POST`** to retrieve data instead of **`GET`**, which is also very common because **`POST`** supports encrypting the request. As you will see in the upcoming sections, Microsoft Visual Studio provides full support to creating, developing, and publishing Web API services. Actually, when implementing communication between a service and a client, you need to know the exchange format and implement your architecture accordingly.

## JSON: A standard data exchange format

The **JavaScript Object Notation** (**JSON**) format is the de facto standard for implementing data exchange between Web API services and mobile applications. There are several reasons for its popularity, but the following are certainly the most important:

- It has a very simple structure
- It has no dependencies
- It is basically plain text that can represent complex data
- It is used as a markup language with extremely simple syntax.

For better understanding, suppose you have an app that needs to exchange information about contacts with a Web API service. The **`Contact`** instances are represented by the following C# **`Contact`** class:

```
public class Contact
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public DateTime DateOfBirth { get; set; }
  public int Age { get; set; }
  public bool IsFamilyMember { get; set; }
}
```

In JSON, a contact would be represented as follows:

```
{
  “firstName”: “Alessandro”,
  “lastName”: “Del Sole”,
  “dateOfBirth”: “1977-05-10T00:00:00”,
  “age”: 44,
  “isFamilyMember”: false
}
```

As you can see, a simple object is defined within brackets, and each property/value pair is separated by a colon. Note how numbers are not enclosed in quotes.

**Tip: By convention, property names in JSON follow the Camel-casing notation (where the first letter is lowercase). However, defining property names with the Pascal-case notation (where the first letter is uppercase) is very common, especially among .NET developers who are used to defining properties this way.**

JSON can also represent data collections. The following markup shows an example based on multiple contacts:

```
{
  “contacts”: [
    {
```

```
    “firstName”: “Alessandro”,
    “lastName”: “Del Sole”,
    “dateOfBirth”: “1977-05-10T00:00:00”,
   “age”: 44,
   “isFamilyMember”: false
  },
  {
   “firstName”: “Robert”,
   “lastName”: “White”,
   “dateOfBirth”: “1990-01-01T00:00:00”,
   “age”: 32,
   “isFamilyMember”: true
  }
  ]
 }
```

Any JSON markup must be enclosed between brackets. Square parentheses represent an array, and individual objects (enclosed between brackets) are enclosed inside the array and separated by a comma. You could have a root object with its own properties and a collection. For example, consider the following C# code:

```
public class People
{
 public List<Contact> Contacts { get; set; }
 public int ID { get; set; }
 public string Owner { get; set; }
}
It is represented by the following JSON:
{
 “contacts”: [
  {
    “firstName”: “Alessandro”,
    “lastName”: “Del Sole”,
    “dateOfBirth”: “1977-05-10T00:00:00”,
   “age”: 44,
   “isFamilyMember”: false
  },
  {
   “firstName”: “Robert”,
   “lastName”: “White”,
   “dateOfBirth”: “1990-01-01T00:00:00”,
   “age”: 32,
   “isFamilyMember”: true
  }
  ],
  “id”: 0,
  “owner”: “System administrator”
}
```

In the previous examples, the JSON markup represents a number of objects of the same type, but you can have different objects under the same node. JSON is not only used to format the data response that a Web API service returns to clients but also to package the request information from the client that makes an API call to the service. This is better explained in the upcoming sections with the actual implementation of Web API, but for now, what you just need to know is that requests from client to service will contain the following headers:

```
Content-Type: application/json
Accept: application/json
```

When clients receive data from a Web API, they need to convert the resulting JSON into C# objects. Luckily enough, there are libraries that allow you to do this very quickly. This is also explained in the upcoming sections.

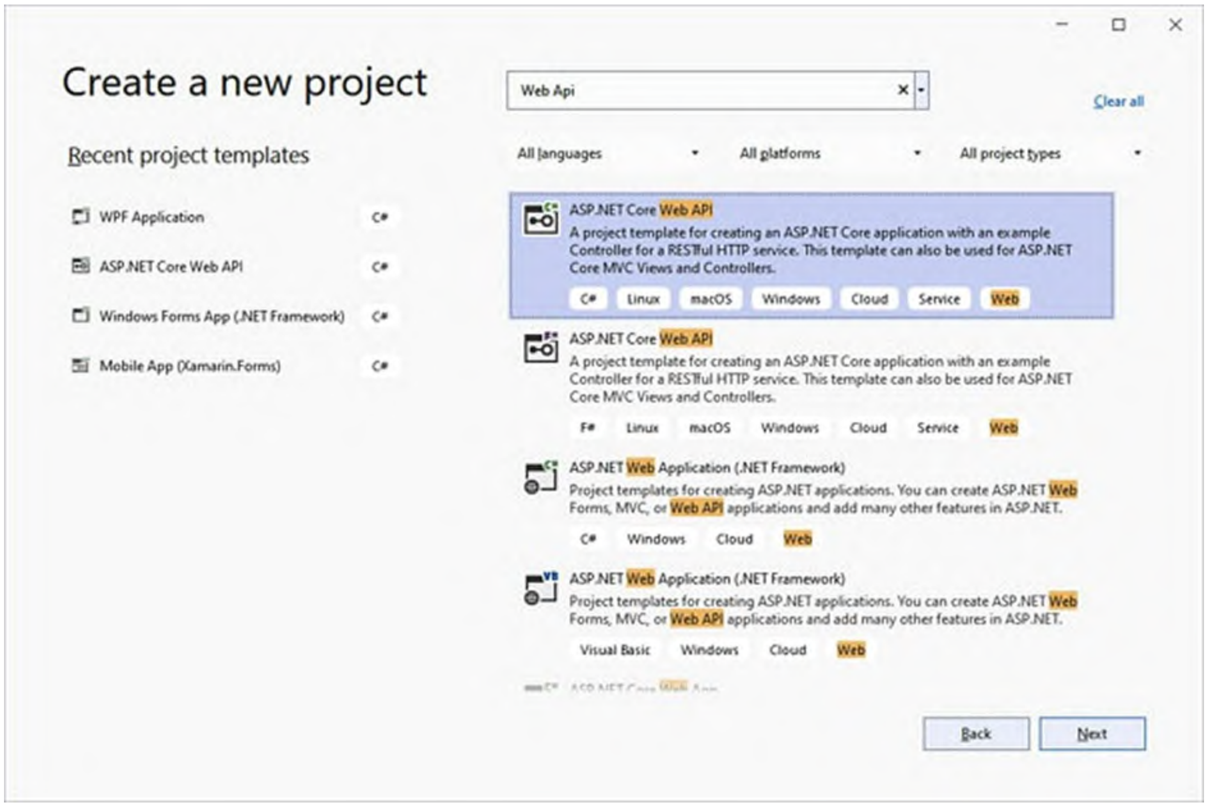
## Creating Web API in Visual Studio

You will now learn how to create a Web API service with C# and ASP.NET Core, and in the next sections, you will learn how to consume this service from a Xamarin.Forms solution. The sample Web API will simulate a remote bookshelf, where you read book information from and where you can add new books. As you can imagine, it is not possible to summarize the ASP.NET Core and Web API technologies in one chapter, so the explanation will focus on the most important concepts about development of Web API services with Visual Studio. For consistency with the Microsoft documentation about consuming Web API from Xamarin.Forms (**https://docs.microsoft.com/en-us/aspnet/core/mobile/native-mobile-backend**) and in order to provide the most abstract implementation possible, data will be made available through an in-memory collection instead of a database.

**In the real world, companies use relational databases to store remote information, such as Microsoft SQL Server, MySQL, PostgreSQL, or SQL on Azure. It is not possible to predict here all the possible scenarios, so the data store will really depend on the company you join. However, the way data is exchanged will be the same. For more information, you can read about Microsoft Entity Framework, the technology used to access databases in a .NET-oriented way**

**(https://docs.microsoft.com/en-us/ef/).**

When ready, in Visual Studio, open the **`Create a new project`** dialog and locate the **`ASP.NET Core Web API project`** template, as shown in *Figure 12.1*. You can help yourself by typing Web API in the search box:



ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web

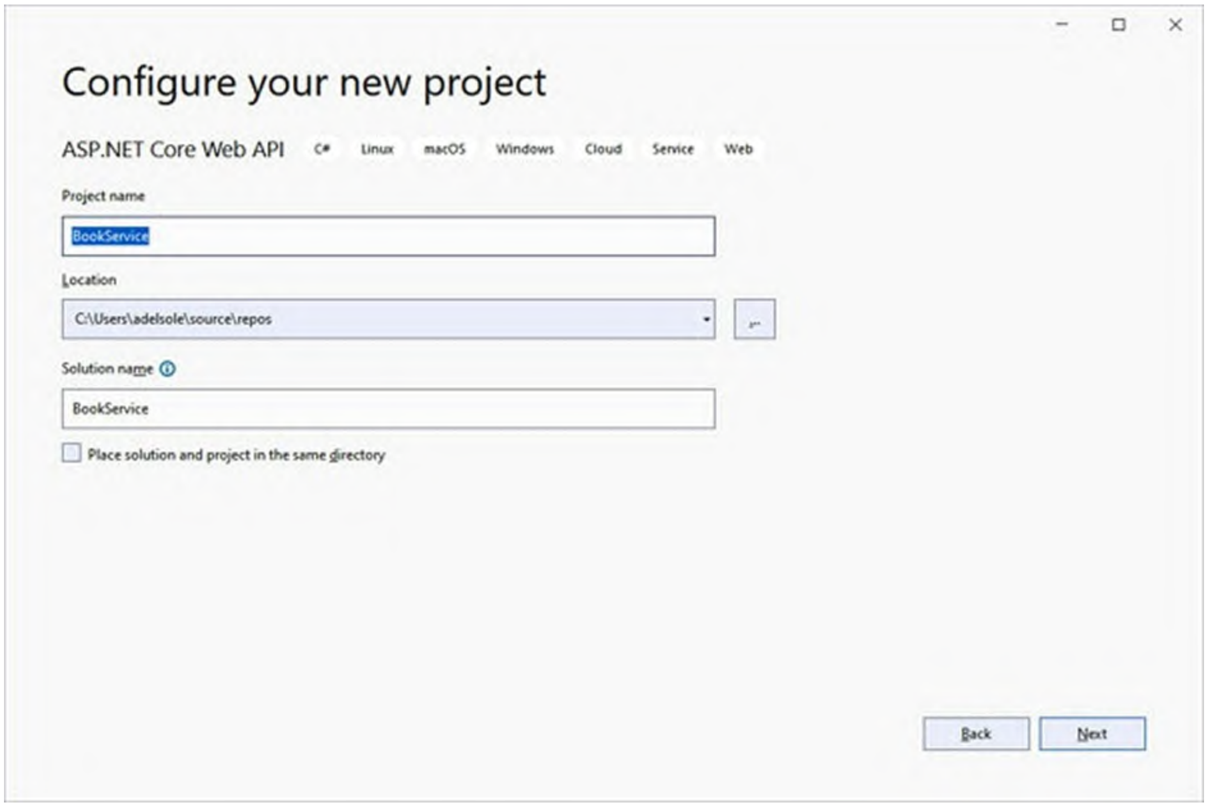
ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
F# Linux macOS Windows Cloud Service Web

ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
C# Windows Cloud Web

ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
Visual Basic Windows Cloud Web

***Figure 12.1:*** *Creating a new Web API project*

When done, click on **`Next`**. In the next screen, specify **`BookService`** as the name for the new project (see *Figure 12.2*) and then click on **`Next`** again:


# Configure your new project

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web

Project name

BookService

Location

C:\Users\adelsole\source\repos

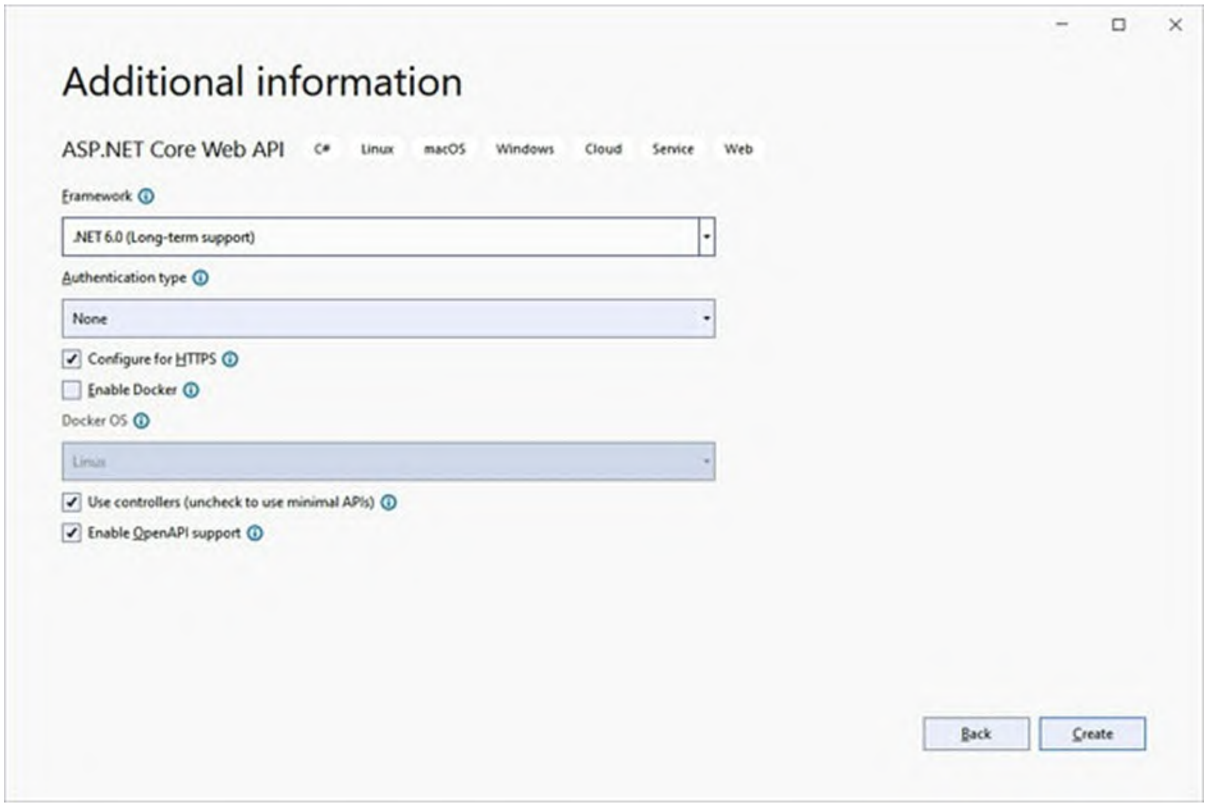
Solution name

BookService

Place solution and project in the same directory

***Figure 12.2:*** *Specifying the project name*

In the **`Additional information`** dialog (see *Figure 12.3*), you will see how Visual Studio proposes the latest version of .NET installed on your machine as the Framework. If you have .NET 6 installed, make sure this is the version selected, or make sure the highest version possible is selected:



Additional information

ASP.NET Core Web API

Framework: .NET 6.0 (Long-term support)

Authentication type: None

[✓] Configure for HTTPS

[ ] Enable Docker

Docker OS: Linux

[✓] Use controllers (uncheck to use minimal APIs)

[✓] Enable OpenAPI support

***Figure 12.3:*** *Specifying project properties*

In the **`Authentication`** type field, you can specify one of the following authentication modes:

- **None**: This is the anonymous authentication type and allows clients to access information without authentication.
- **Microsoft identity platform**: This is an evolution of the Microsoft Active Directory authentication technology, and it is based on the OAuth 2.0 standard.
- **Windows**: This is used within a Windows domain and allows for authenticating clients via the user credentials used to log in to Windows.

Like for the data store, for the sake of simplicity and because it is not possible to predict any real scenario you will work on, the *None authentication* is used here. Many companies also implement their own authentication mechanism based on username and password, which is not uncommon at all. Leave all the options as they are and click on **`Create. After a few seconds`**, the project will be ready. Visual Studio automatically generates code that simulates a weather forecast service. This code will not be used in this chapter so that you can learn step-by-step how to implement the necessary components manually, but it is not necessary that you remove any file.

## Understanding the project structure

There are some relevant points that you need to know about the structure of Web API projects. The first point is controllers. A controller is a class that can be considered as a container of API calls exposed to the public, where API calls are .NET methods that can be invoked from client applications and that return a result. When you create a Web API project, a default sample controller called **`WeatherForecastController`** is added to the **`WeatherForecastController.cs`** file. You will learn more about controllers when creating a new one, but what you need to know now is that controllers contain the methods you want clients to be able to invoke. The second point you need to know is about is the configuration information of the service, which is stored in two JSON files: **`appsettings.json`** and **`launchSettings.json`**.

**JSON is now widely used also to define projects' configuration, especially in the latest versions of Visual Studio, replacing XML. So, it is not only used for data exchange, but also for local object definition.**

Both files are influenced by the way you set project properties, but when a new project is created, they have a standard structure. The **`appsettings.json`** file determines how the runtime flow is traced by the logging system, whereas the **`launchSettings.json`** file stores the startup and publishing information set up in the project properties. It is not necessary to change them directly because you will work with the dedicated Visual Studio windows. What you need to know is that those files are the physical places where Visual Studio stores some configuration. Looking at their content is left to your choice. The next point to discuss is the **`Program.cs`** file.

**With .NET 6, the `Program.cs` file contains code that is implicitly defined inside a `Program` class. This means that there is no class `Program { }` block, but only the body of the class. With lower versions, the class `Program { }` block is still defined explicitly.**

This file contains the **`Program`** class and all the necessary startup code and, when a new project is created, has the following content:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
  app.UseSwagger();
  app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

The builder variable is of type **`WebApplicationBuilder`** and is responsible for defining the services exposed by the Web API via a collection called **`Services`**, of type **`IServiceCollection`**. Its **`AddControllers`** method adds the controllers found in the project to the list of exposed services; the **`AddEndpointsApiExplorer`** method exposes metadata that contain the list of controllers and other services so that clients can know what a service offers; the **`AddSwaggerGen`** method adds support for Swagger (**https://swagger.io/**), a tool that automates the generation of Web API documentation. Swagger is not part of .NET and is automatically added to the solution by Visual Studio as a NuGet package. You will see the result produced by Swagger in the section called **Testing the Web API Locally**. The app variable is of type **`WebApplication`** and represents a running instance of the Web API service. This is an important point: a Web API service is a static service, which means that an instance is created only when it is invoked. The next lines of code detect if the service is running on a development machine, like yours; if so, it enables the service to display the documentation generated by Swagger. Next, it enables the so-called HTTPS redirection via the **`UseHttpsRedirection`** method. This is important, especially when debugging, because local communications happen via HTTP and not HTTPS, so this mechanism makes sure HTTP calls are interpreted as HTTPS. The last two methods, **`MapControllers`** and **`Run`**, finalize the exposure of controllers and start an instance of the Web API service, respectively. You will still work with the **`Program.cs`** file. In fact, you will need to specify the instance of the class that will handle your data as one of the services. This is done in the next section.

## Creating a data model

The goal of the sample project is to expose calls that make it possible to work with a virtual **`bookshelf`**, so you need a class that represents an individual book. It can be very simple, like the following definition:

```
public class Book
{
  public int ID { get; set; }
  public string Title { get; set; }
  public DateTime PublicationDate { get; set; }
  public string ISBN { get; set; }
}
```

The **`ID`** property is important because it allows for uniquely identifying an object and because it is going to be used by the API methods to retrieve such unique objects. You now need a service that can handle the data. The Web API projects require you to first define an interface, called service, that establishes a set of methods and properties to work with data. The following code shows how to define a service interface to work with books:

```
public interface IBookRepository
{
  bool DoesItemExist(int id);
  IEnumerable<Book> All { get; }
  Book Find(int id);
  void Insert(Book item);
  void Update(Book item);
  void Delete(int id);
}
```

**Tip: The `Repository` suffix is not mandatory, but it is used for consistency with project examples in the official Web API documentation.**

Member names are self-explanatory: **`DoesItemExist`** checks if a **`Book`** object already exists given the id; all is a property that returns the full list of **`Book`** objects; find returns a specific instance given the id; and **`Insert`**, **`Update`**, and **`Delete`** allow for adding, modifying, and removing an instance of the **`Book`** class. At this point, you need a class that implements the **`IBookRepository`** interface. In the real world, such a class would implement methods that access a database, but in this example, it will work with a sample in-memory collection. It is called **`BookRepository`** and is defined as follows:

```
public class BookRepository : IBookRepository
{
  private List<Book> Books;
  public IEnumerable<Book> All => Books;
  public BookRepository()
  {
   InitializeData();
  }
  public void Delete(int id)
  {
   Books.Remove(this.Find(id));
  }
  public bool DoesItemExist(int id)
  {
   return Books.Any(item => item.ID == id);
  }
  public Book Find(int id)
  {
   return Books.FirstOrDefault(item => item.ID == id);
  }
  public void Insert(Book item)
  {
   Books.Add(item);
  }
  public void Update(Book item)
  {
   var book = this.Find(item.ID);
   var index = Books.IndexOf(book);
   Books.RemoveAt(index);
   Books.Insert(index, item);
  }
  private void InitializeData()
  {
   Books = new List<Book>();
   var book1 = new Book
   {
    ID = 1,
    ISBN = “9789391392871”,
    Title = “Practitioner’s Guide to Data Science”,
    PublicationDate = new DateTime(2022, 1, 1)
   };
   var book2 = new Book
```

```
    {
     ID = 2,
     ISBN = “9789355510068”,
     Title = “IoT for Beginners”,
     PublicationDate = new DateTime(2021, 12, 1)
    };
    var book3 = new Book
    {
     ID = 3,
     ISBN = “9789355511102”,
     Title = “iOS 15 Application Development for Beginners”,
     PublicationDate = new DateTime(2021, 12, 1)
    };
    Books.Add(book1);
    Books.Add(book2);
    Books.Add(book3);
   }
 }
```

The previous code uses techniques you already know. It invokes the **`Add`** and **`Remove`** methods over the **`Books`** collection to insert and delete **`Book`** instances, and it invokes the **`FirstOrDefault`** method to retrieve a specific **`Book`** instance given the id. The **`All`** property simply returns the content of the **`Books`** collection, whereas the **`Update`** method first removes the existing instance of the **`Book`** object from the collection and adds a new one containing any edits. The **`Insert`**, **`Update`**, and **`Delete`** methods will be mapped to the **`POST`**, **`PUT`**, and **`DELETE`** HTTP verbs in a controller, and the **`All`** property will be mapped to the **`GET`** verb. The **`InitializeData`** method generates some sample data, simulating a database table.

**Tip: In the real world, it is not recommended to physically delete an item from a database (or another data source). Instead, a better approach is to add a bool column to the database called `IsDeleted` (or similar) and set its value to true when an object is deleted. This will also require you to change the method that returns the list of data by filtering out those whose `IsDeleted` value is true.**

Before the data service can be used, it needs to be added to the collection of services exposed by the Web API. To accomplish this, you need to add the line of code highlighted in bold to the **`Program.cs`** file:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddSingleton<IBookRepository, BookRepository>
```

```
 ();
builder.Services.AddControllers();
```

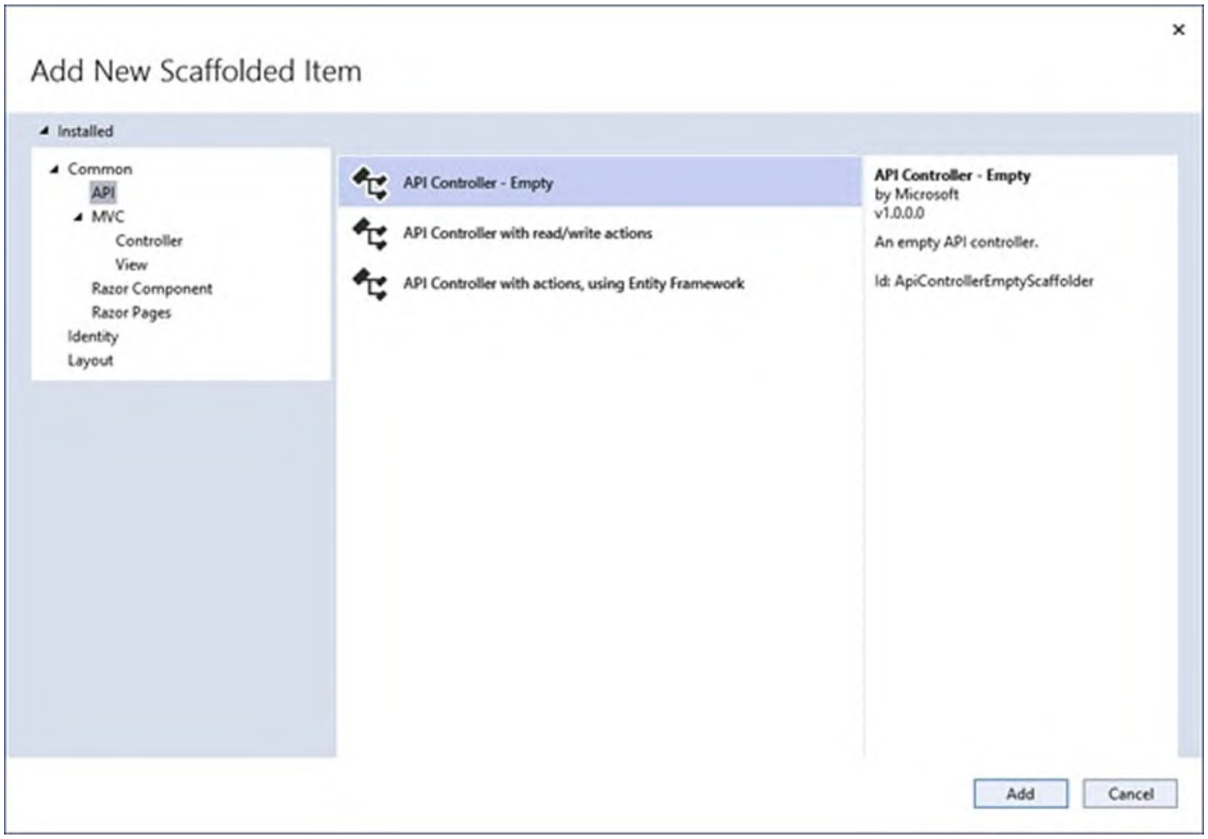
The **`AddSingleton`** methods adds a unique instance of the specified data service. You need to specify the service interface and one specialized implementation.

## Implementing controllers

The next step is to implement methods that clients can invoke to perform operations against the data, and this must be done inside a controller. Visual Studio offers tools that simplify the generation of controller classes. To accomplish this, right-click on the project name in **`Solution Explorer`** and select **`Add`** | **`New Scaffolded Item`**.

**Tip: In the Visual Studio terminology, scaffolding means generating code files that are already set up to work by including the necessary references, namespaces, and minimal code.**

In the **`Add New Scaffolded Item`** dialog (see *Figure 12.4*), select the **`API Controller – Empty item`** and then click on **`Add`**.



API Controller - Empty
by Microsoft
v1.0.0.0
An empty API controller.
Id: ApiControllerEmptyScaffolder

***Figure 12.4:*** *Adding a new empty API controller*

At this point, you will be able to specify a name for the controller file (see *Figure 12.5*), so enter **`BooksController.cs`** and finally, click on **`Add`**:

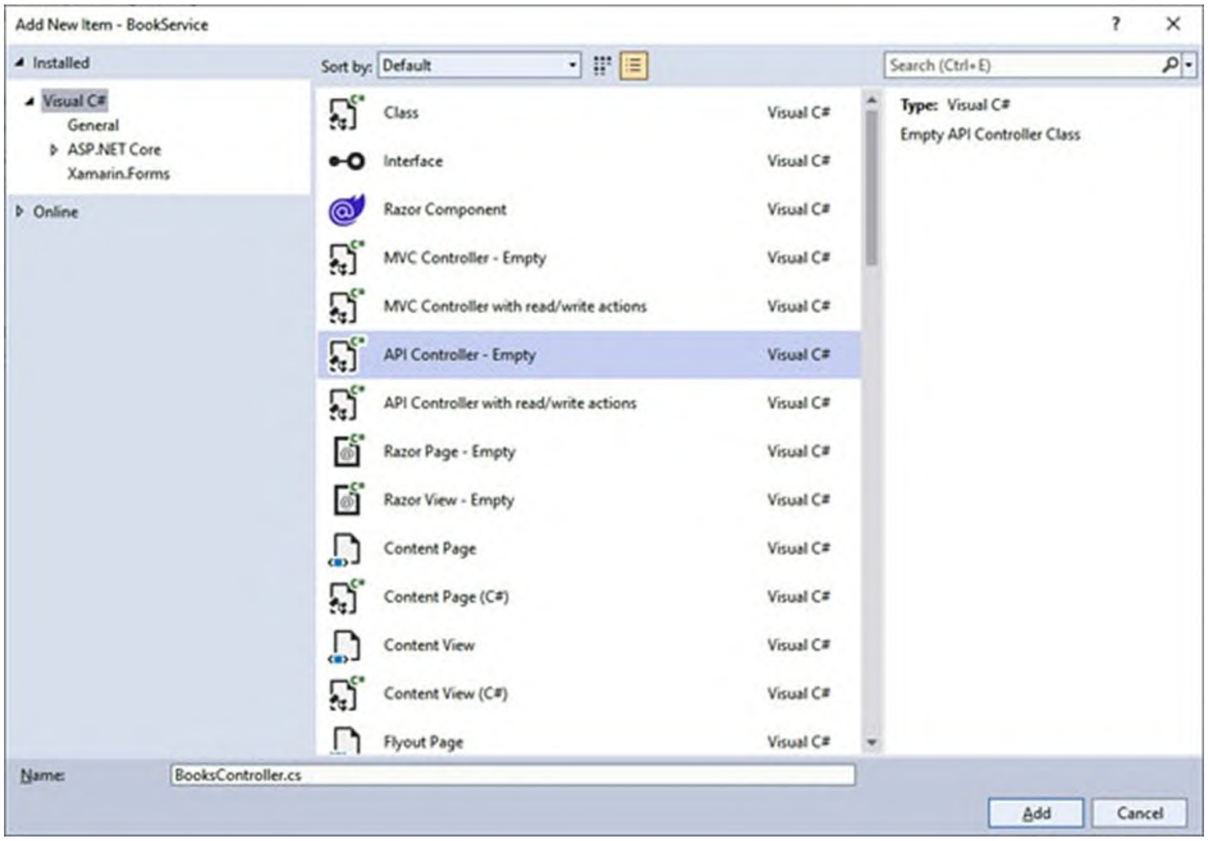

Add New Item - BookService
Installed
Visual C#
General
ASP.NET Core
Xamarin.Forms
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C#
Interface Visual C#
Razor Component Visual C#
MVC Controller - Empty Visual C#
MVC Controller with read/write actions Visual C#
API Controller - Empty Visual C#
API Controller with read/write actions Visual C#
Razor Page - Empty Visual C#
Razor View - Empty Visual C#
Content Page Visual C#
Content Page (C#) Visual C#
Content View Visual C#
Content View (C#) Visual C#
Flyout Page Visual C#
Type: Visual C#
Empty API Controller Class
Name: BooksController.cs
Add
Cancel

***Figure 12.5:*** *Specifying the controller's name*

When the new code file is ready, you will first notice the following class definition:

```
[Route("api/[controller]")]
[ApiController]
public class BooksController : ControllerBase
{
…
}
```

The **`Route`** attribute is of extreme importance because it defines the URL for the controller. In this case, this controller will be invoked as follows:

```
https://serverName/BookService/api/Books
```

The **`Route`** attribute is adding the **`api`** suffix to the service URL, and it makes it possible to invoke the controller by only specifying the name, without the **`Controller`** suffix. This will be clearer in the next section, when testing the service. The **`ApiController`** attribute makes sure that .NET recognizes the class as a Web API controller, and the inheritance from **`ControllerBase`** provides specific objects for controllers, which are discussed in the upcoming section.

## Retrieving data

The first code you need to add is the class's constructor, plus a field that stores the instance of the data context. You can write it as follows:

```
private readonly IBookRepository bookRepository;
public BooksController(IBookRepository _bookRepository)
{
 bookRepository = _bookRepository;
}
```

When the service starts up, the **`AddSingleton`** method you saw in the **`Program`** class will also invoke the constructor of the controller, passing the instance of the data context object. Now, you can write the simple code that returns the full list of books:

```
[HttpGet]
public IActionResult List()
{
 try
 {
  return Ok(bookRepository.All);
 }
 catch (Exception)
 {
  return BadRequest(ErrorCodes.CouldNotGetItems.ToString());
 }
}
```

The name of the method, **`List`** in this case, is not relevant because the service will intercept the **`GET`** verb specified with the **`HttpGet`** attribute, regardless of the method name. **`List`** returns an object of type **`IActionResult`**, which specifies a successful call along with data, or an error state. **`Ok`** is a method that returns a successful status code, and all the objects in the **`bookRepository`** instance. In case of error, the code returns a **`Bad Request`** status code via the **`BadRequest`** method. This method takes a string as a parameter, in this case, the result of the invocation of **`ToString`**, over values of an enumeration called **`ErrorCodes`**, defined as follows:

```
public enum ErrorCodes
{
 InvalidBookData,
 BookIDInUse,
 RecordNotFound,
 CouldNotCreateItem,
 CouldNotUpdateItem,
 CouldNotDeleteItem,
  CouldNotGetItems
}
```

This enumeration is completely custom and allows for sending your own error codes.

## Creating and updating data objects

The next step is to write code that adds and updates Book objects. The first method you add is called create, and it maps the **POST** verb as follows:

```
[HttpPost]
public IActionResult Create([FromBody] Book item)
{
  try
  {
   if (item == null || !ModelState.IsValid)
   {
    return BadRequest(ErrorCodes.InvalidBookData.ToString());
   }
   bool itemExists = bookRepository.DoesItemExist(item.ID);
   if (itemExists)
   {
    return StatusCode(StatusCodes.Status409Conflict,
       ErrorCodes.BookIDInUse.ToString());
   }
   bookRepository.Insert(item);
  }
  catch (Exception)
  {
   return BadRequest(ErrorCodes.CouldNotCreateItem.ToString());
  }
  return Ok(item);
}
```

The first thing to underline is that the method receives the **Book** object to add as a parameter, which is also decorated with the **FromBody** attribute. This means that the object is inside the body of the HTTP request and .NET will decode the data from there. The code is then quite simple to understand: if the object is null or the data model is not in a valid state (**!ModelState.IsValid**), a **BadRequest** is sent; if an object with the same **ID** already exists in the collection, the **StatusCode** method is invoked to send a HTTP error with code **409**, which means data conflict (**StatusCodes.Status409Conflict**); if everything is fine, the Insert method is invoked to add the new object to the collection.

**Tip: Remember that, in the real-world, checks over data and the invocation to methods like `Insert` would be done against a data access layer, like the Entity Framework, instead of an in-memory collection.**

When updating existing items, you could write the following method that maps the **`PUT`** verb:

```
[HttpPut]
public IActionResult Edit([FromBody] Book item)
{
 try
 {
  if (item == null || !ModelState.IsValid)
  {
   return BadRequest(ErrorCodes.InvalidBookData.ToString());
  }
  var existingItem = bookRepository.Find(item.ID);
  if (existingItem == null)
  {
   return NotFound(ErrorCodes.RecordNotFound.ToString());
  }
  bookRepository.Update(item);
 }
 catch (Exception)
 {
  return BadRequest(ErrorCodes.CouldNotUpdateItem.ToString());
 }
 return NoContent();
}
```

The difference with the previous method here is that the code checks if an item with the specified ID exists and, if so, it invokes the **`Update`** method of the data service; otherwise, it sends an error code.

## Deleting data objects

The last part of the controller is related to implementing an action that deletes the specified object from the data context and that maps the **`DELETE`** verb. The following code demonstrates this:

```
[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
 try
 {
  var item = bookRepository.Find(id);
```

```
    if (item == null)
    {
     return NotFound(ErrorCodes.RecordNotFound.ToString());
    }
    bookRepository.Delete(id);
   }
   catch (Exception)
   {
    return BadRequest(ErrorCodes.CouldNotDeleteItem.ToString());
   }
   return NoContent();
  }
```

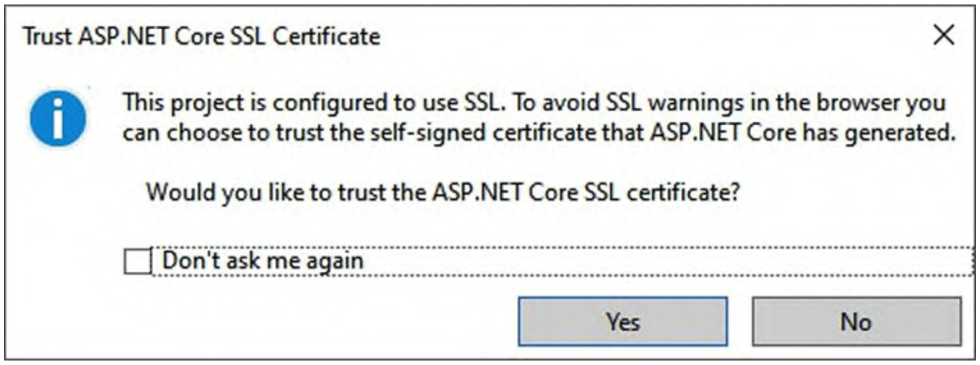
You might have immediately noticed a couple of differences with the previous methods. First, the **`HttpDelete`** attribute requires the specification of an ID. This means that the **`ID`** of the object is sent along with the URL of the invocation, like this:

```
https://serverName/BookService/api/Books/1
```

Here, 1 is a sample ID. The second difference is that the method parameter is no longer passed via **`FromBody`**; instead, it is taken from the URL of the invocation. If an object with the specified ID is found, the **`Delete`** method from the data service class is invoked. Now that you have written all the necessary code, you can test the service locally before publishing it to a remote server.

## Testing Web API services

As you would do with any other application, the Web API service must be debugged and tested before it is released to the general audience. Testing and debugging the service on your development machine is very easy because the Visual Studio Installer also sets up an execution environment called Internet Information Services Express (or simply, IIS Express), which allows you to run Web API as well as other web applications. IIS Express is not discussed here in detail, so you can check the documentation (**https://docs.microsoft.com/en-us/iis/extensions/introduction-to-iis-express/iis-express-overview**) for more information. You can start the service by simply pressing *F5*. If you have never launched a similar kind of application, you will be first asked permission to trust an auto-generated SSL certificate that is necessary to work locally (see *Figure 12.6*):

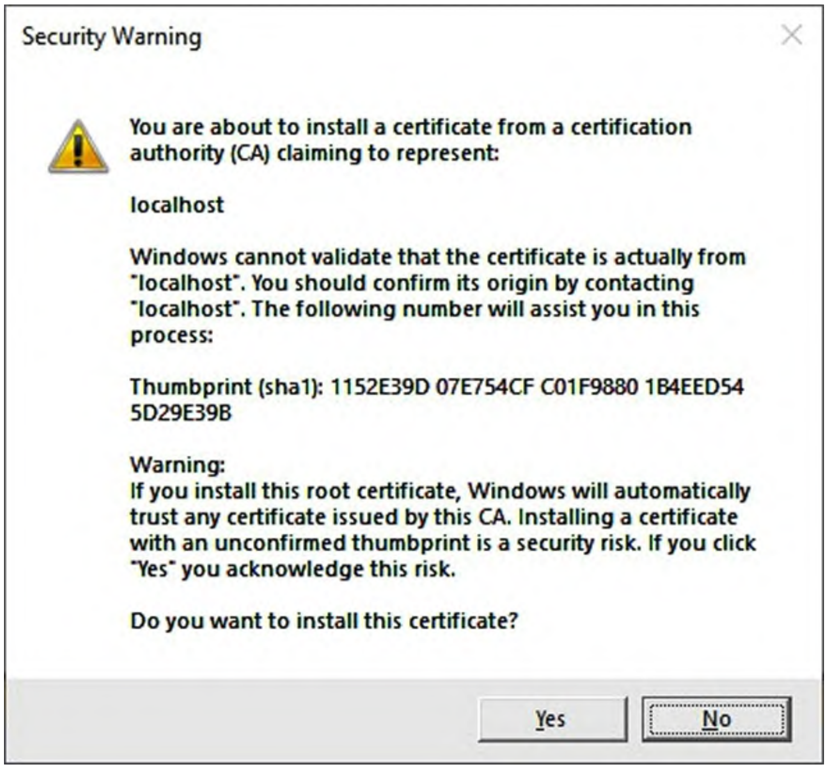

Trust ASP.NET Core SSL Certificate

This project is configured to use SSL. To avoid SSL warnings in the browser you can choose to trust the self-signed certificate that ASP.NET Core has generated.

Would you like to trust the ASP.NET Core SSL certificate?

***Figure 12.6:*** *Trusting an auto-generated SSL certificate*

You can safely click on `Yes`. You will then be asked for a final confirmation about installing the auto-generated certificate into the local list of trusted authorities, as shown in *Figure 12.7*. You can also accept this:


Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

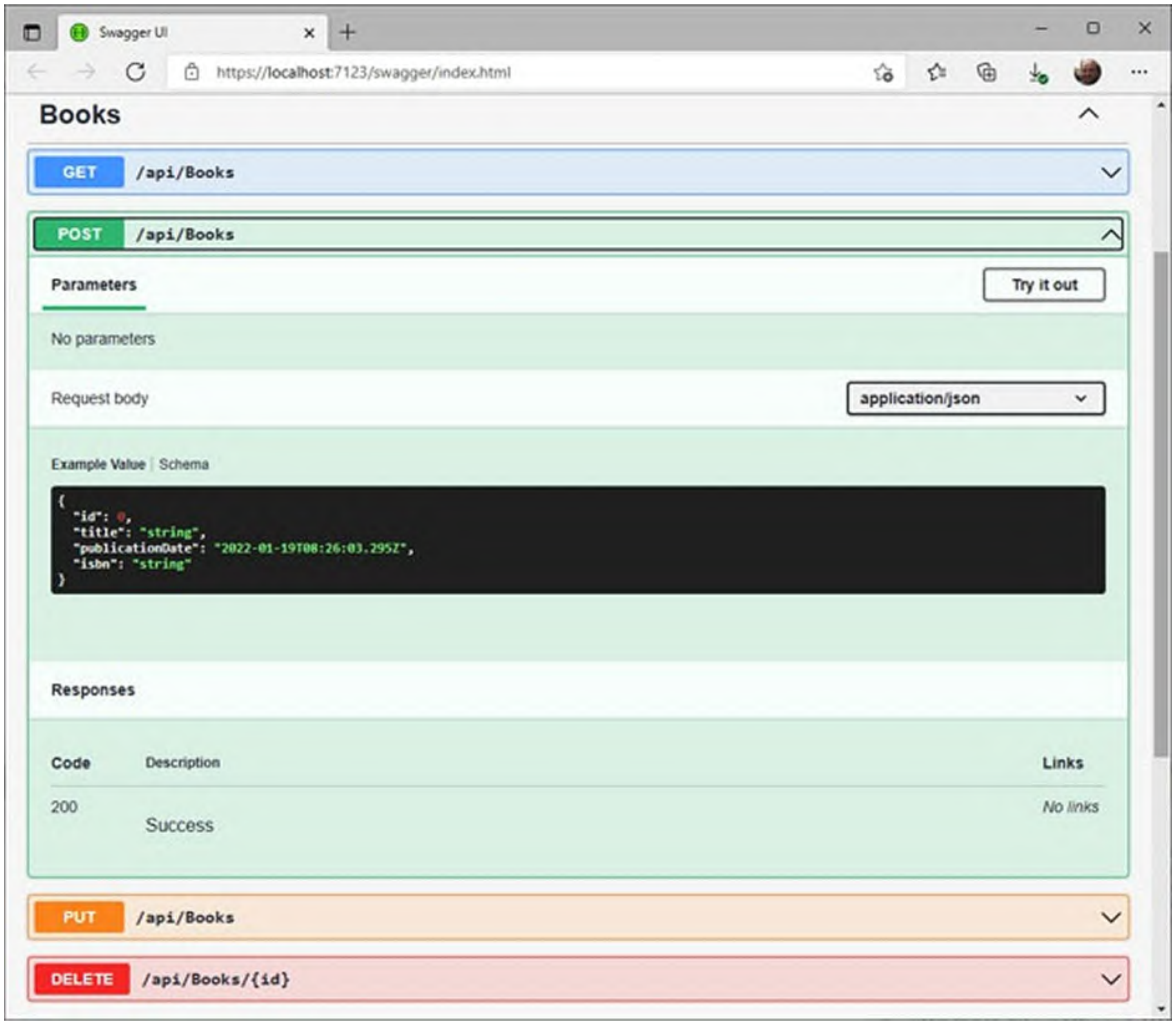
Thumbprint (sha1): 1152E39D 07E754CF C01F9880 1B4EED54 5D29E39B

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

***Figure 12.7:*** *Installing the auto-generated SSL certificate*

After the certificate is installed, your Web API service will start up, and your default browser will show the Swagger user interface, as shown in *Figure 12.8*:



```
{
  "id": 0,
  "title": "string",
  "publicationDate": "2022-01-19T08:26:03.295Z",
  "isbn": "string"
}
```

| Code | Description | Links |
|---|---|---|
| 200 | Success | No links |

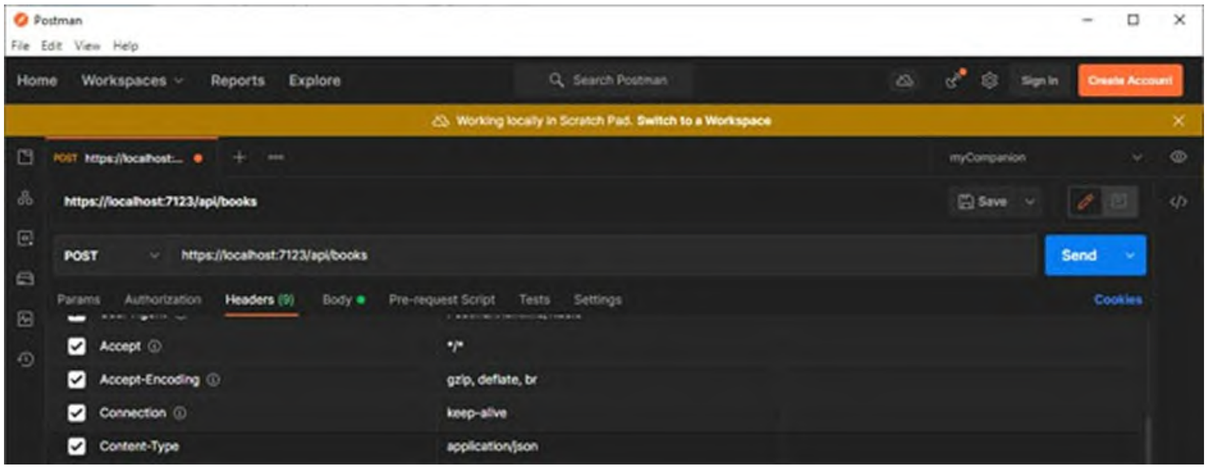
***Figure 12.8:*** *Installing the auto-generated SSL certificate*

The addition of Swagger to the Web API projects is quite recent, and it is a very nice addition because this library automatically generates the documentation of your Web API with a very nice user interface. You can browse API calls, see their parameters, try the API directly by clicking on the **`Try it out`** button close to each call, and enter the appropriate parameters where necessary. While the included tools work very well to test the API calls, most companies use more advanced testing and debugging tools, like Postman, which you downloaded and installed previously.

## Making API calls with Postman

Postman is a very powerful tool and, in this chapter, you will use its most common features to support API calls with JSON requests and responses.

When you open Postman, the first thing you need to do is configure the request headers to include support for JSON content. Click on the **`Headers`** tab (see *Figure 12.9*), scroll the list of pre-populated headers and add a new one called **`Content-Type`** (a built-in word completion engine will help you here). The value for this header is application/json:



https://localhost:7123/api/books

POST https://localhost:7123/api/books

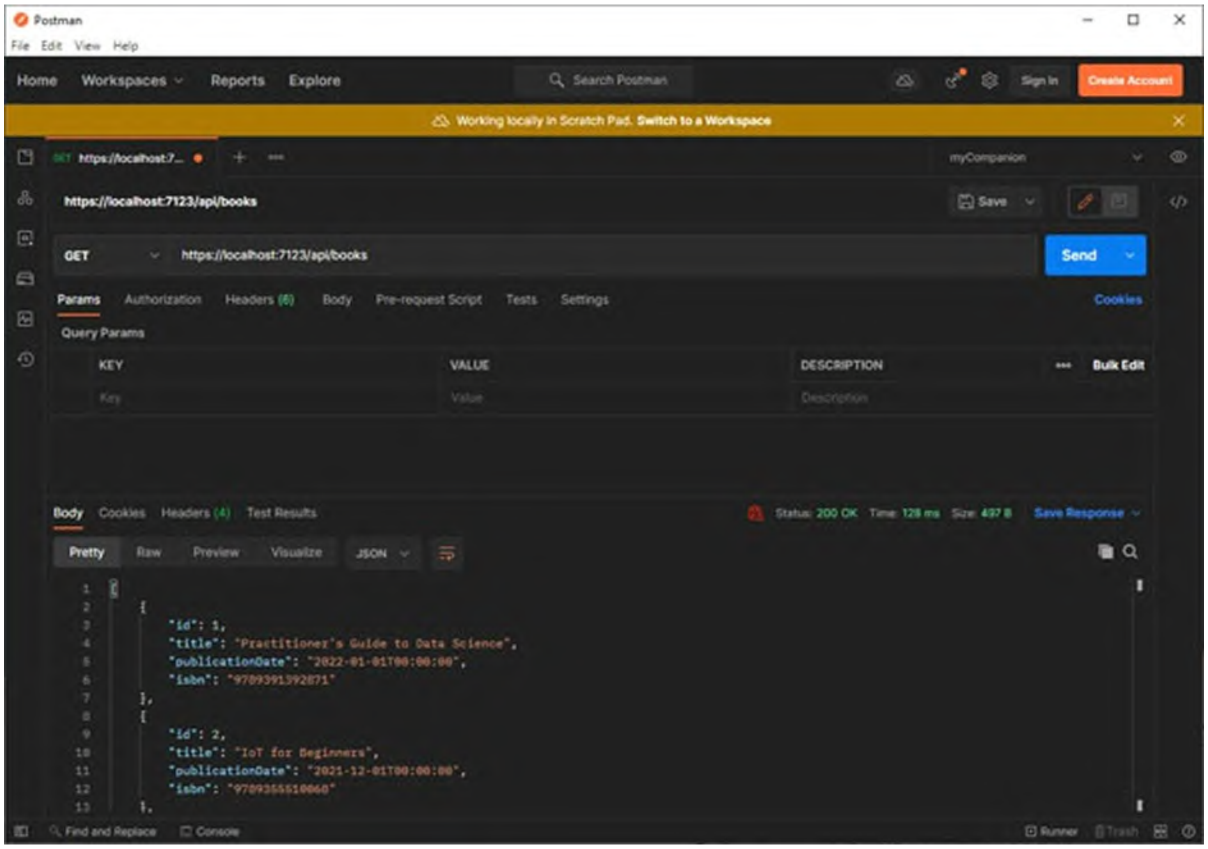
| | |
|---|---|
| ✓ Accept | */* |
| ✓ Accept-Encoding | gzip, deflate, br |
| ✓ Connection | keep-alive |
| ✓ Content-Type | application/json |

***Figure 12.9:*** *Setting up Postman*

Once you have done this, in the address bar, write the URL of the Web API service, which can be taken from your browser's address bar, followed by the **`api/Books`** suffix. You might recall how this suffix was set up by the **`Route`** attribute in the **`BooksController`** class definition.

**Tip: Your Web API service is hosted on the local server, called localhost, at the address specified by the port number specified after the colon. In the current example, it is `https://localhost:7123`. You can change it by selecting `Debug` | `ProjectName Debug Properties` and then locating the `App URL` field of the `Launch Profiles` dialog.**

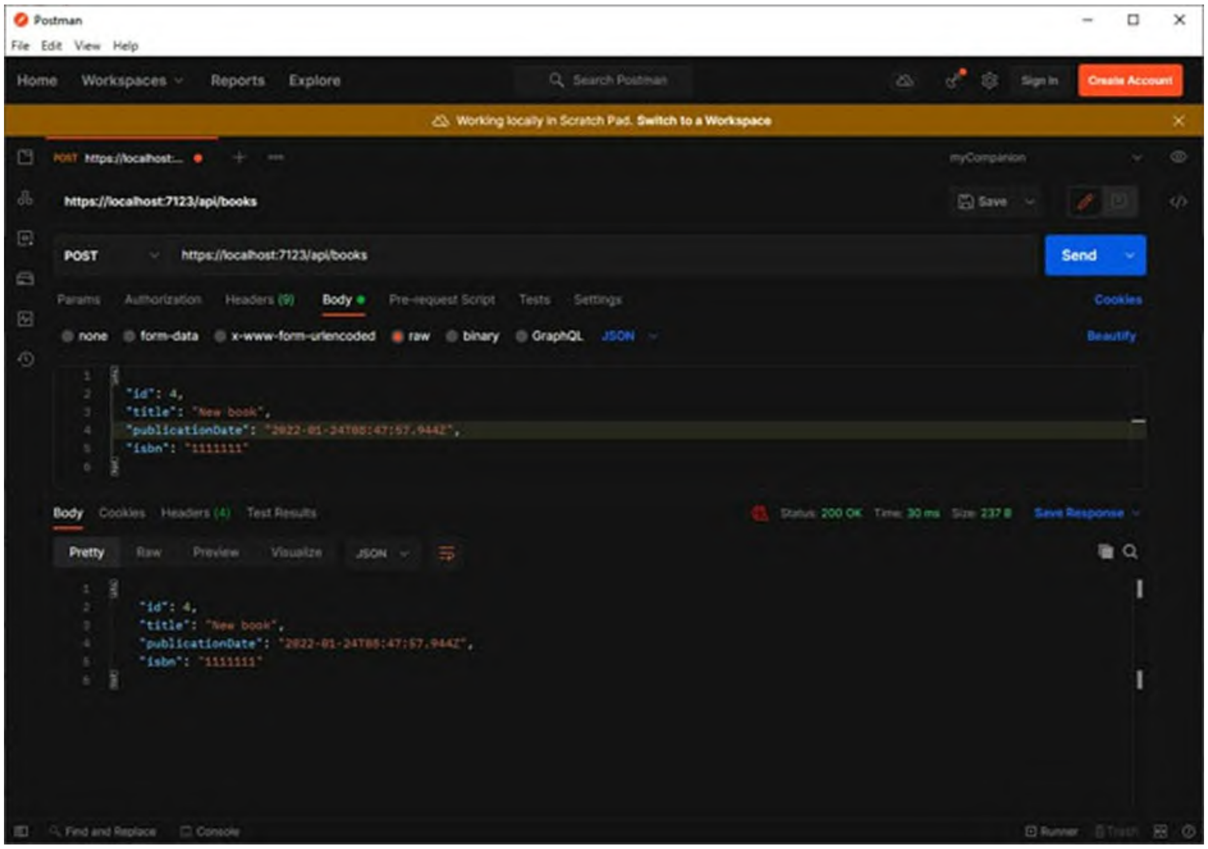
Make sure the **`GET`** verb is selected on the left side of the address bar, as shown in *Figure 12.10*, and then click on **`Send`**. At the bottom of the window, you will see the list of **`Book`** objects in JSON format (see *Figure 12.10*). Additionally, note how Postman shows the **`HTTP`** status code (**`Status 200 OK`**) and the time in milliseconds required to receive the request and send a response:


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Sign In Create Account
Working locally in Scratch Pad. Switch to a Workspace
GET https://localhost:7...
myCompanion
https://localhost:7123/api/books
Save
GET
https://localhost:7123/api/books
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
Query Params
KEY VALUE DESCRIPTION
Bulk Edit
Key Value Description
Body Cookies Headers (4) Test Results
Status: 200 OK Time: 128 ms Size: 497 B
Save Response
Pretty Raw Preview Visualize JSON
Find and Replace Console
Runner Trash

```
[
    {
        "id": 1,
        "title": "Practitioner's Guide to Data Science",
        "publicationDate": "2022-01-01T00:00:00",
        "isbn": "9789391392871"
    },
    {
        "id": 2,
        "title": "IoT for Beginners",
        "publicationDate": "2021-12-01T00:00:00",
        "isbn": "9789355510060"
    },
```

***Figure 12.10:*** *Retrieving data with a GET call*

You do not need anything else because the Web API service is simply responding to a `HTTP GET` request. If you needed to pass parameters, for example, to filter a query result, you could use the same approach used over the `DELETE` verb. Writing new data requires a `POST` call, so make sure you change from `GET` to `POST` in the combo box at the left side of the address bar. You also need to pass the information of a new `Book` object. To accomplish this, click on the `Body` tab (see *Figure 12.11*) and then select the raw content type. This allows for providing the object information directly in JSON. You could choose a different way, but this is out of scope here. Provide the new object information as you like, and then click on `Send`. If everything goes fine, the Web API service returns an instance of the new object, which is displayed at the bottom of the window:


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Sign In
Create Account
Working locally in Scratch Pad. Switch to a Workspace
POST https://localhost:...
myCompanion
Save
Send
Params Authorization Headers (9) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (4) Test Results
Save Response
Pretty Raw Preview Visualize JSON
Find and Replace Console
Runner Trash

https://localhost:7123/api/books

POST https://localhost:7123/api/books

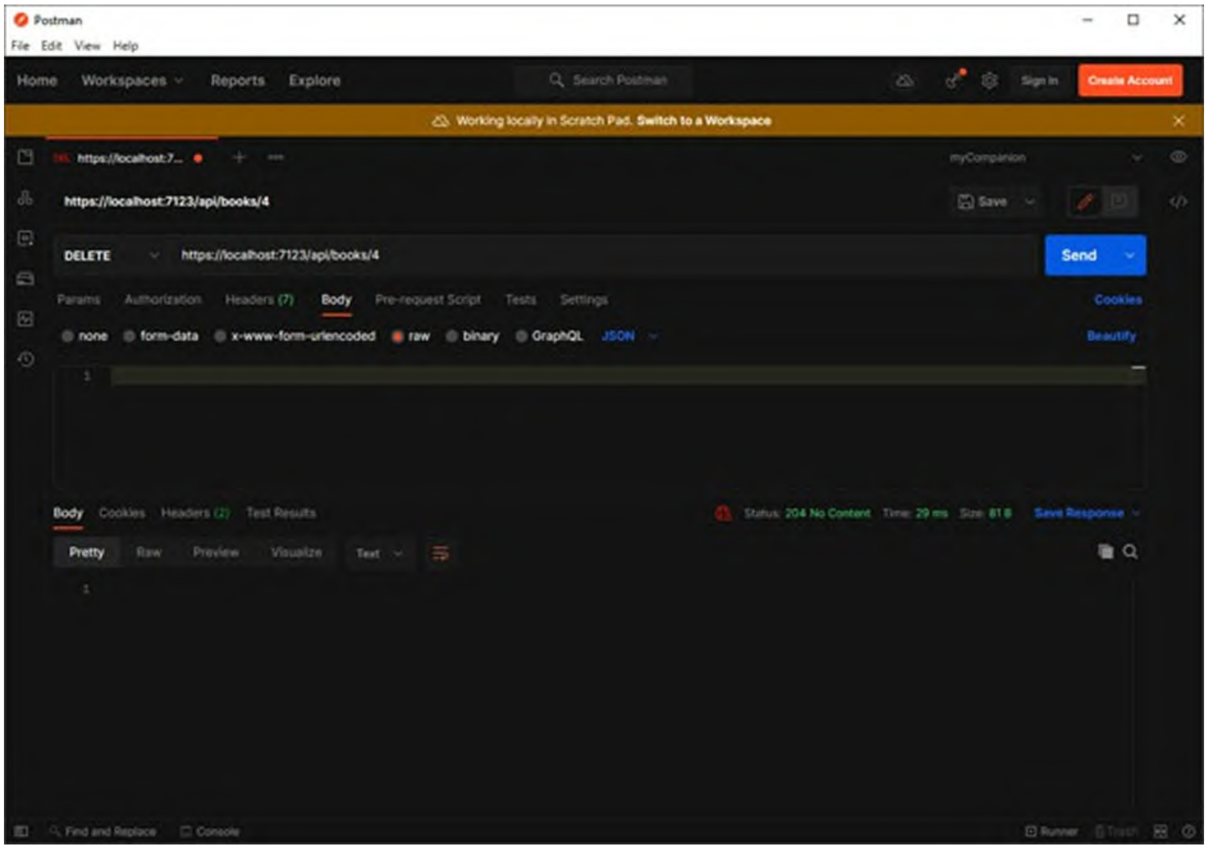
```
{
  "id": 4,
  "title": "New book",
  "publicationDate": "2022-01-24T08:47:57.944Z",
  "isbn": "1111111"
}
```

Status: 200 OK Time: 30 ms Size: 237 B

```
{
    "id": 4,
    "title": "New book",
    "publicationDate": "2022-01-24T08:47:57.944Z",
    "isbn": "1111111"
}
```

***Figure 12.11:*** *Writing data with a POST call*

If you wish to update an existing object, change from **`POST`** to **`PUT`** as the HTTP verb and in the JSON of the object, first specify the **`ID`** of an existing item and then change the other properties as desired; finally, click on **`Send`**. The last API call maps the **`DELETE`** verb, so change the selection from **`PUT`** to **`DELETE`** and then, in the address bar, add a slash followed by the **`ID`** of the object you want to remove (see *Figure 12.12*). If everything works as expected, the Web API service returns a **`204 status`** code (**`No Content`**).


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Sign In
Create Account
Working locally in Scratch Pad. Switch to a Workspace
myCompanion
Save
Send
Params Authorization Headers (7) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (2) Test Results
Save Response
Pretty Raw Preview Visualize Text
Find and Replace Console
Runner Trash

https://localhost:7123/api/books/4

DELETE https://localhost:7123/api/books/4

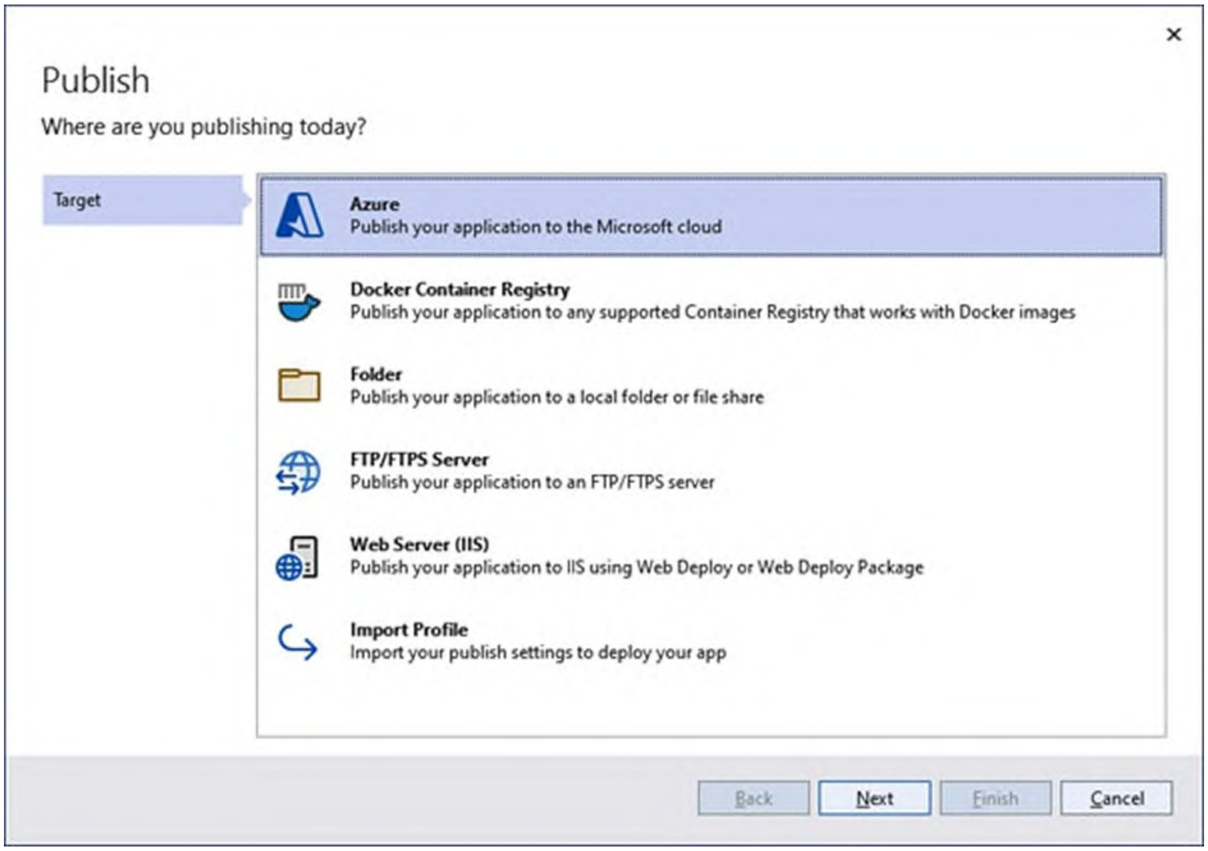
Status: 204 No Content Time: 29 ms Size: 81 B

***Figure 12.12:*** *Deleting items with a DELETE call*

After working with Postman, in the real world, you will likely call API from your mobile app to see if everything works fine and debug it as necessary before you publish everything to a remote server. However, for instructional purposes only, the Web API service will now be first published to Azure and finally, consumed from a Xamarin.Forms project.

## Publishing Web API services

Web API services, and in general, web services and web applications, can be published to different targets such as on-premises servers, cloud platforms, local development servers, shared remote folders, and even to ZIP packages that will later be handled by a system administrator. Visual Studio has integrated support for all these scenarios, so publishing from within the IDE is a simplified task (assuming that you have all the necessary credentials and permissions to publish to the desired target). Assuming that you have already set up an Azure trial subscription, in **`Solution Explorer`**, right-click on the project name and then click on **`Publish`**. When the **`Publish`** dialog appears (see *Figure 12.13*), select **Azure** as the target and click on **Next**.


Publish

Where are you publishing today?

Target

Azure
Publish your application to the Microsoft cloud

Docker Container Registry
Publish your application to any supported Container Registry that works with Docker images

Folder
Publish your application to a local folder or file share

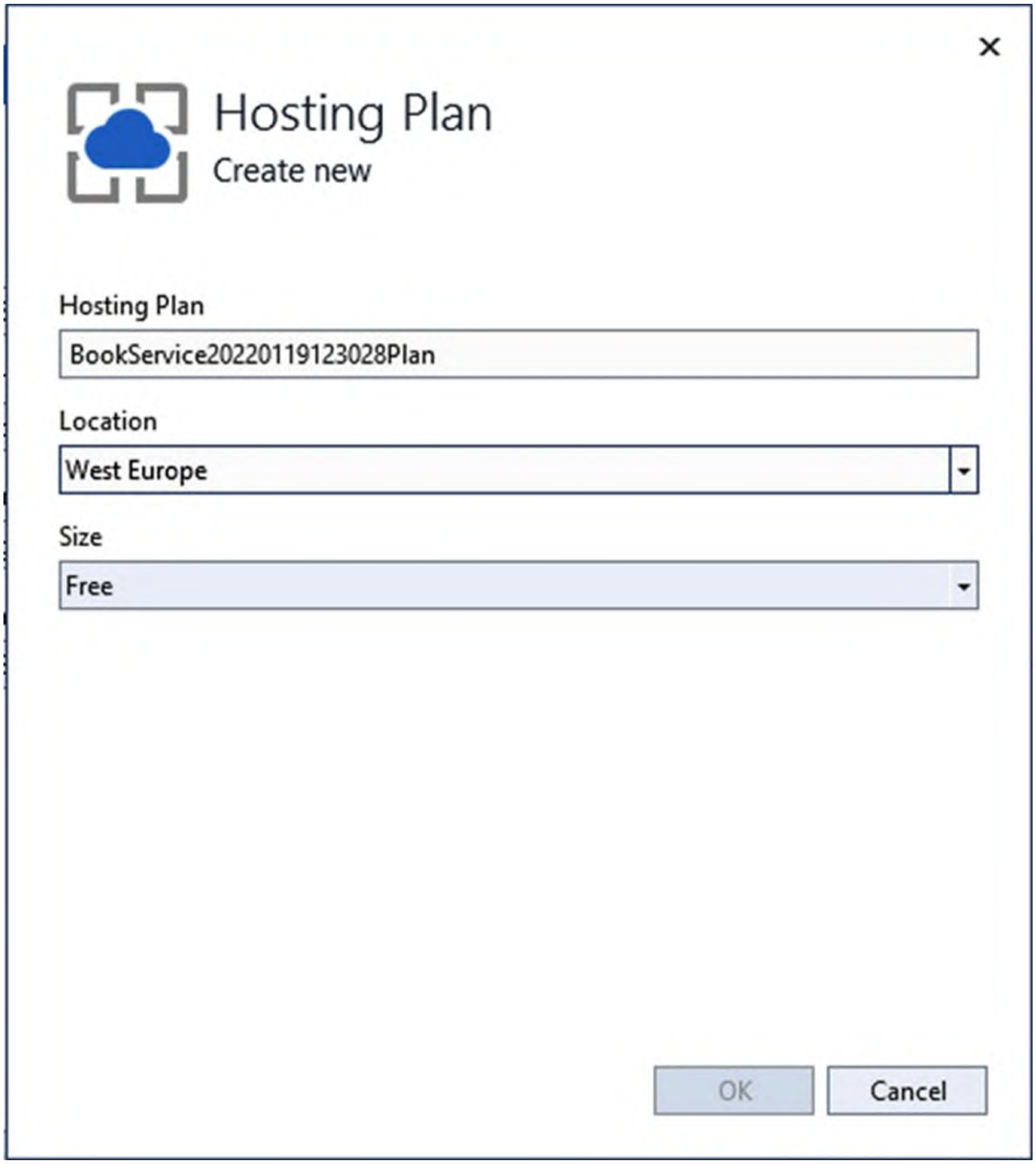
FTP/FTPS Server
Publish your application to an FTP/FTPS server

Web Server (IIS)
Publish your application to IIS using Web Deploy or Web Deploy Package

Import Profile
Import your publish settings to deploy your app

***Figure 12.13:** Selecting a publish profile*

Publishing to Azure requires you to set up two building blocks: an app service, which acts as a container of services, and a Web API service. You might be asked to enter the credentials of the Microsoft account that is associated to your Azure subscription. In the next dialog, select **Azure App Service (Windows)** as the system infrastructure and click on **Next**. The next step is to define a hosting plan, which means choosing a data center region and the size of the service. If you look at *Figure 12.14*, you can see how Visual Studio generates a name for the hosting plan, which you can leave unchanged. The **Location** combo box allows you to select a region. Choose the closest one to your **Country** of residence to reduce bandwidth and latency. In the **Size** combo box, make sure you select the **Free plan**. This plan offers less computational resources than all the other plans, but it is more than enough for the purposes of this chapter. When everything is set up, click on **OK**:


Hosting Plan

Create new

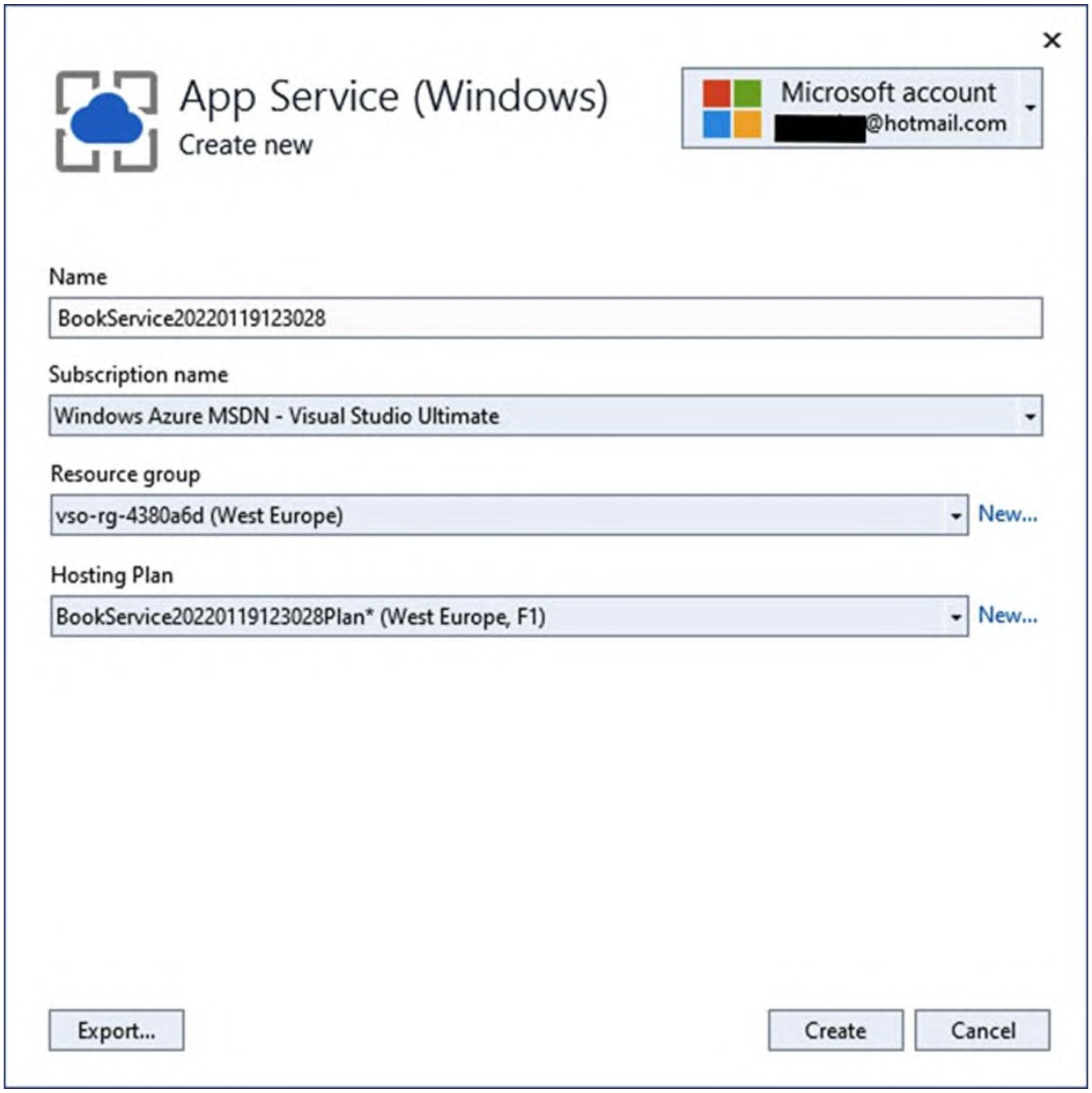
Hosting Plan: BookService20220119123028Plan

Location: West Europe

Size: Free

***Figure 12.14:*** *Selecting a hosting plan*

In the **`App Service (Windows)`** dialog (see *Figure 12.15*), enter a name for the service, or leave unchanged the one proposed by Visual Studio. This will be used to identify the service in the Azure management portal. Make sure that the selected Subscription name is assigned with your subscription and, if no resource group exists, create a new one by clicking on the **`New`** button. In the **`Hosting Plan combo`** box, ensure that the hosting plan created previously is selected. When ready, click on **Create**:


App Service (Windows)
Create new

Microsoft account
[redacted]@hotmail.com

Name
BookService20220119123028

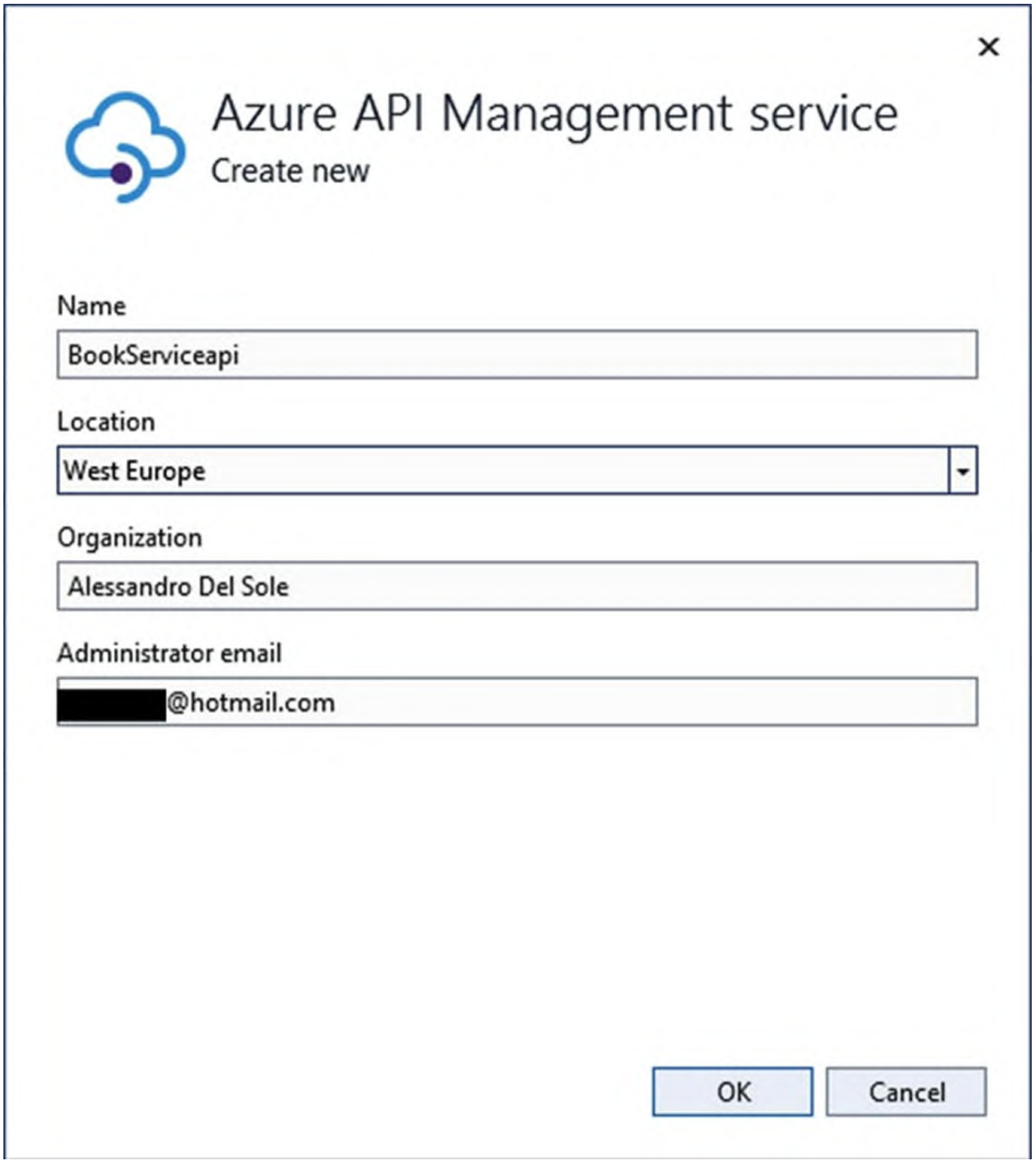
Subscription name
Windows Azure MSDN - Visual Studio Ultimate

Resource group
vso-rg-4380a6d (West Europe)

Hosting Plan
BookService20220119123028Plan* (West Europe, F1)

***Figure 12.15:*** *Configuring Azure service resources*

At this point, you will be prompted with the summary information of the Web API, in particular, the name, as shown in *Figure 12.16*. Visual Studio automatically provides a name, in this case **BookServiceapi**, which is acceptable, so leave it as it is. Ensure that **Location**, **Organization**, and **Administrator email** contain valid values, and then click on **OK**:



Azure API Management service

Create new

Name

BookServiceapi

Location

West Europe

Organization

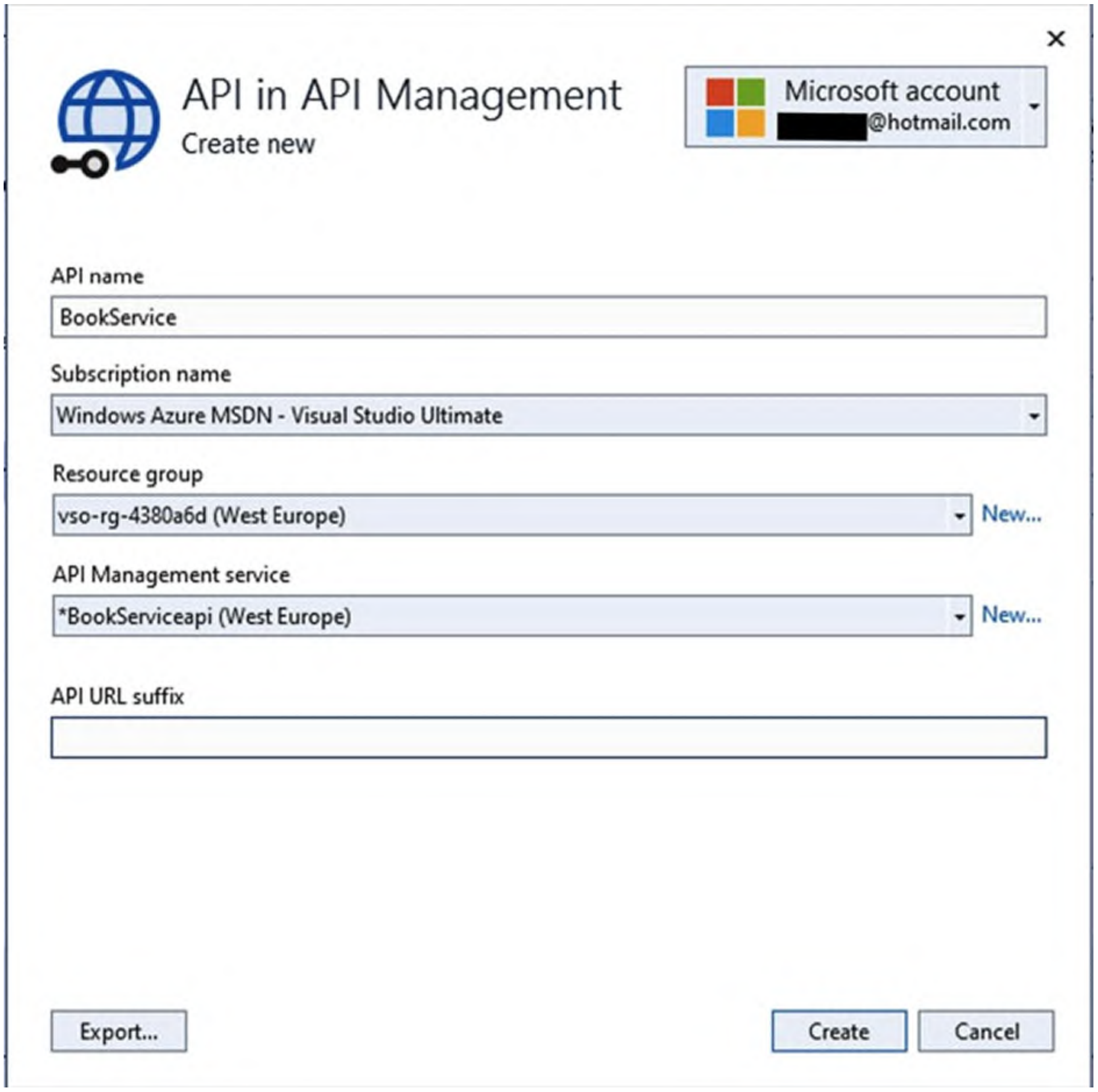
Alessandro Del Sole

Administrator email

[redacted]@hotmail.com

***Figure 12.16:*** *Setting up the Web API name*

In the last step, a summary of the Web API service is provided, as shown in *Figure 12.17*. Check that every field is assigned with what you have decided previously and, when ready, click on **`Create`**:



API in API Management

Create new

Microsoft account

[redacted]@hotmail.com

API name

BookService

Subscription name

Windows Azure MSDN - Visual Studio Ultimate

Resource group

vso-rg-4380a6d (West Europe)

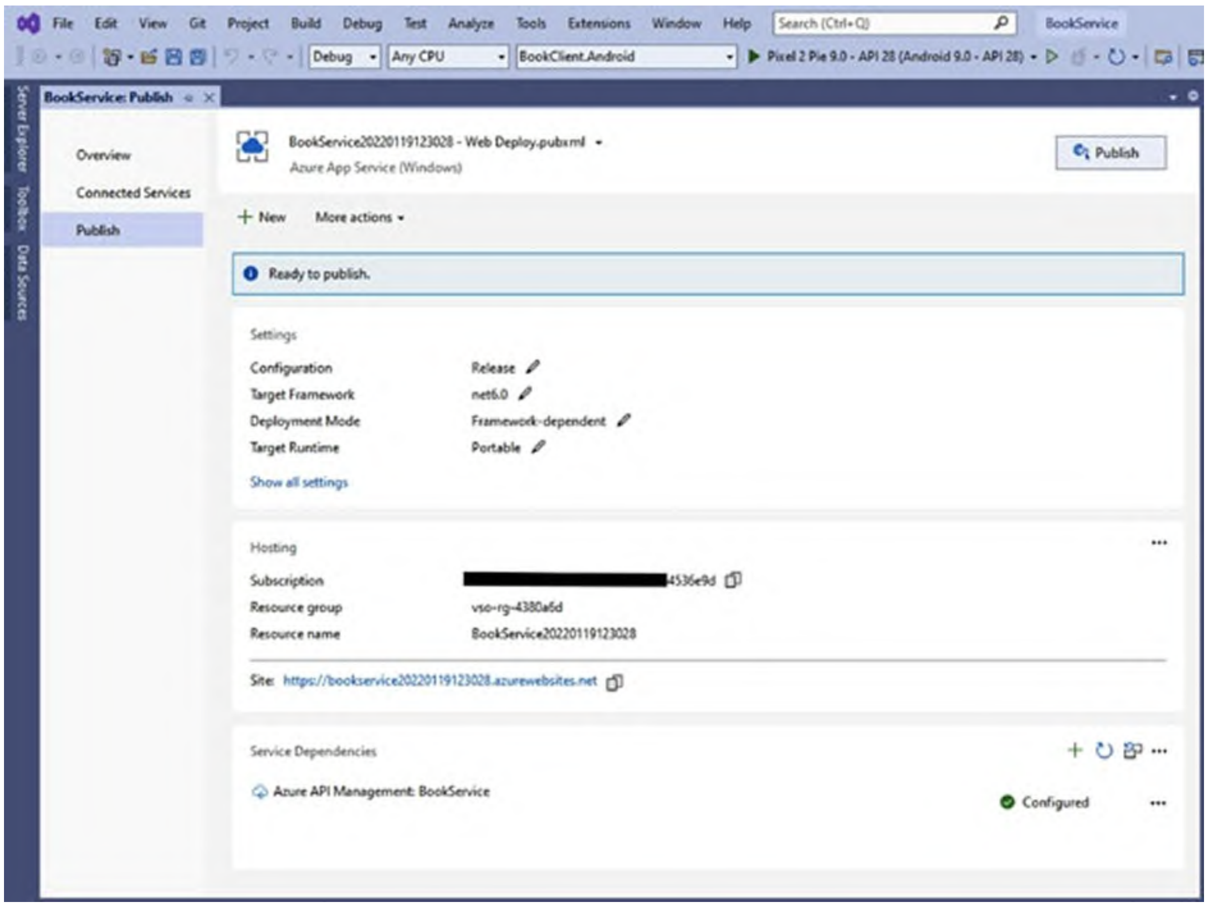
API Management service

*BookServiceapi (West Europe)

API URL suffix

***Figure 12.17:*** *Configuring Web API properties*

When the dialog closes, Visual Studio generates a new publish profile that can also be reused later and offers the **`Publish`** button (see *Figure 12.18*). When you click on it, Visual Studio starts packaging and publishing the Web API service to your **`Azure subscription`**. The time required for publication depends on the complexity of your project, so in this case, it should be very quick



BookService20220119123028 - Web Deploy.pubxml

Azure App Service (Windows)

Ready to publish.

Settings

Configuration Release

Target Framework net6.0

Deployment Mode Framework-dependent

Target Runtime Portable

Show all settings

Hosting

Subscription [redacted]4536e9d

Resource group vso-rg-4380a6d

Resource name BookService20220119123028

Site: https://bookservice20220119123028.azurewebsites.net

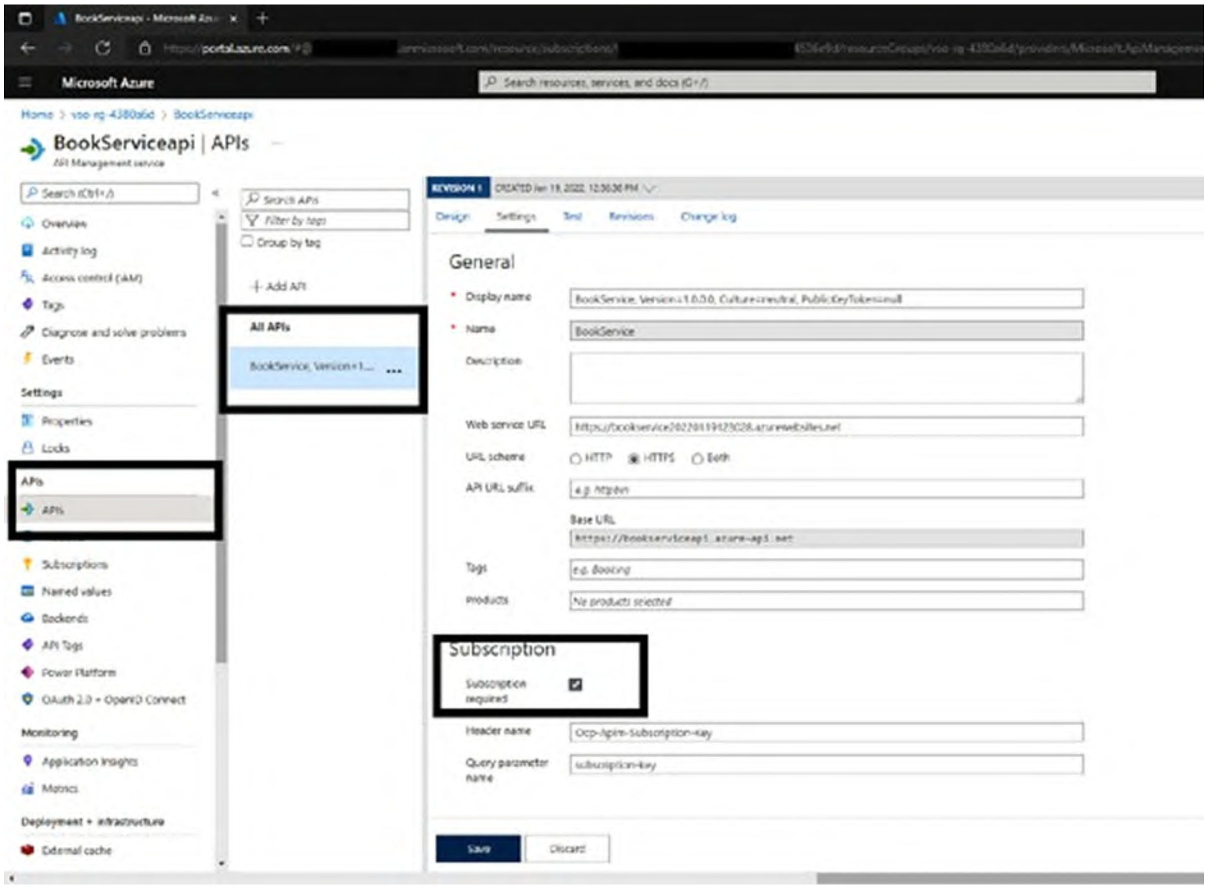
Service Dependencies

Azure API Management: BookService Configured

***Figure 12.18:*** *Reviewing publish options*

When the publication process is completed, Visual Studio will open an instance of your default web browser pointing to the address of your Web API service. However, this will likely result in a **`404 – Not found error`** because the sample service has no user interface defined and because, by default, Azure automatically configures Web API services so that any call should include a subscription key for security reasons. The current sample service is based on anonymous authentication, so you need to disable the inclusion of a subscription key; otherwise, no client will be able to invoke the API (unless you obviously include the subscription key). To accomplish this, open the Azure Management Portal website (**https://portal.azure.com**) and sign in with your credentials. When the portal appears, click on the **`BookServiceapi`** resource (or the name you previously specified) in the list of most recent resources. This will open a management page for the Web API. On the left side of the screen, click on **`APIs`** in the toolbar and then click on the service name in the column placed between the tool bar and the details' view (see *Figure 12.19*). When the details appear in the right-side of the screen, click on the **`Settings`** tab and, at the bottom of the page, locate and unselect the **`Subscription`** required field.

**Tip: You can always include a subscription key for more secure Web API access by including the `Ocp-Apim-Subscription-Key` header in the API request, like you did for the `Content-Type` header in Postman, passing the subscription key that can be retrieved by clicking on `Subscriptions` in the toolbar. You will get an example of adding headers in C# in the next section.**



General

Display name: BookService, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Name: BookService
Description:
Web service URL: https://bookservice20220119123028.azurewebsites.net
URL scheme: ○ HTTP ● HTTPS ○ Both
API URL suffix: e.g. httpbin
Base URL: https://bookserviceapi.azure-api.net
Tags: e.g. Booking
Products: No products selected

Subscription

Subscription required: ✓
Header name: Ocp-Apim-Subscription-Key
Query parameter name: subscription-key

***Figure 12.19:*** *Accessing Web API properties on Azure*

When ready, click ib **`Save`**. Keep note of the service URL, which you can find in the Web service URL field. This will be used shortly. You can close the Management Portal if you wish. Before consuming the service from a Xamarin.Forms project, it is a good idea to test it from Postman to make sure everything is set up properly. To accomplish this, follow these steps:

1. Open Postman.

2. In the address bar, replace the local address of the service with the one you copied from the Azure management portal.
3. Repeat the steps described in the previous section to make API calls to read, insert, and delete data.

When you have checked that everything is working as expected, you are ready to move to the last part of the exercise: consuming the remote service from a Xamarin.Forms project.

**Even if Azure subscriptions have a spending limit enabled by default, and even if you selected a free hosting plan, it is recommended to delete resources you no longer use, especially when you are experimenting, in order to avoid the risk of charging your bank account. Shutting down services is not enough; you need to fully delete unused resources and services. This might be the case of the current sample service.**

## Consuming Web API with Xamarin.Forms

In this section, you will learn how to invoke Web API services from Xamarin.Forms and interact with data from your native mobile apps. Most of your gained knowledge will be reused here, so we will focus only on new concepts. You can add a new Xamarin.Forms project to the existing solution so that you have all the sample code in one place. To accomplish this, in **`Solution Explorer`**, right-click on the solution name and then click on **`Add`** | **`New Project`**. By reusing your existing skills, create a new Xamarin.Forms project called **`BookClient`**. When the project is ready, the first thing to do is to add support for JSON to .NET and C#. The latest versions of .NET Core include native support for JSON via a namespace called **`System.Text.Json`**, but this is not available in Xamarin.Forms (it is instead available in .NET MAUI). Moreover, there is a library called **`Newtonsoft.Json`** which has become the de-facto standard over the years to work against JSON data, so you will learn how to work with it, becoming ready to also work on existing projects that leverage this library. In **`Solution Explorer`**, right-click on the **`BookClient`** project name (only the shared project) and then select Manage NuGet Packages. When the NuGet user interface is shown, you will see the **`Newtonsoft.Json`** package already in the list of the most popular libraries (see *Figure 12.20*). Select the package and click on **`Install`**. Installing the package will only take a few seconds:

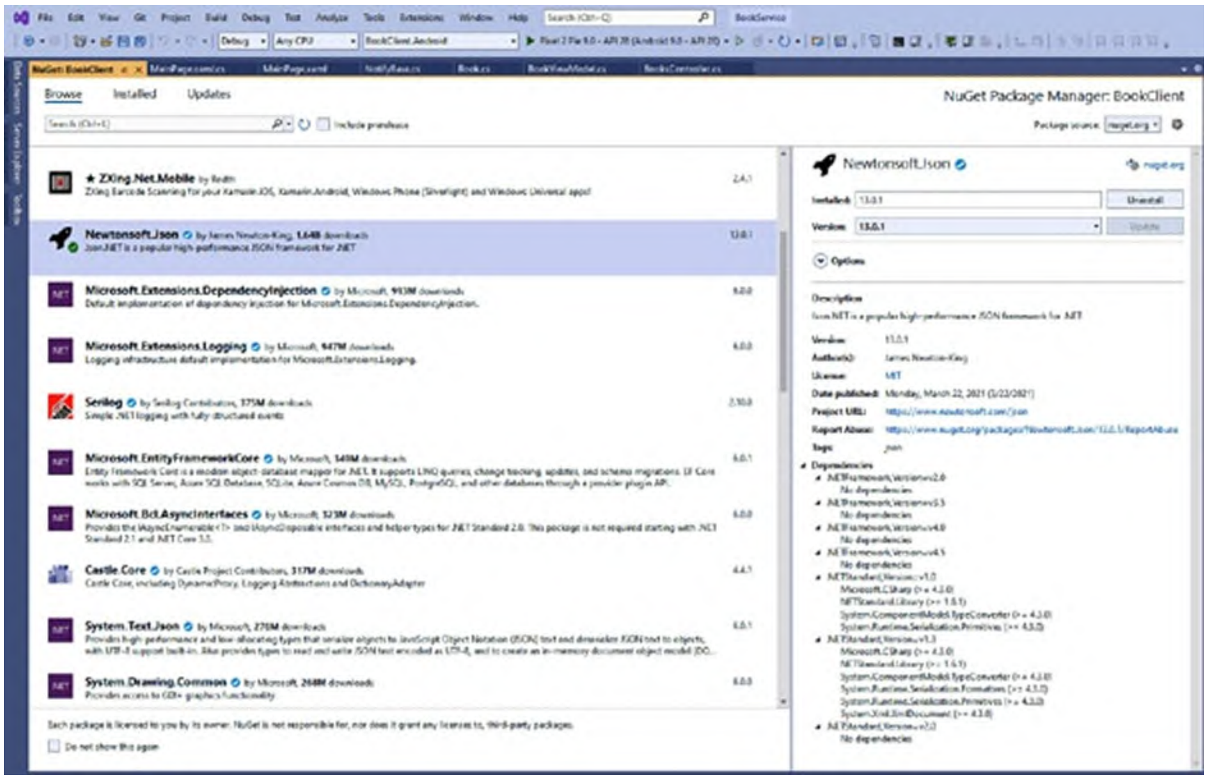


*Figure 12.20: Installing the Newtonsoft.Json package*

Now that you have added support for JSON data, you can create the necessary infrastructure based on the Model-View-ViewModel pattern.

## Creating a data model

In terms of code, the first thing you need is a class that represents a book. However, both this class and the viewmodel that you will create in the next section need to send change notifications, so it is a good idea to create a base class that implements the **`INotifyPropertyChanged`** interface and that will be used by both the model and the viewmodel. You can call the class **`NotifyBase`** and define it as follows:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace BookClient.Model
{
  public class NotifyBase : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged([CallerMemberName] string
    propertyName = null)
    {
     PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

At this point, you can define a class called **`Book`**, which has the same properties of the **`Book`** class defined in the Web API service, but with the addition of property change notification, so that the user interface will be able to automatically refresh according to changes. The following is the simple code for the **`Book`** class:

```
 using System;
 namespace BookClient.Model
 {
  public class Book : NotifyBase
  {
   private int _id;
   public int ID
   {
    get { return _id; }
    set { _id = value; OnPropertyChanged(); }
   }
   private string _title;
   public string Title
   {
    get { return _title; }
    set
    {
     _title = value; OnPropertyChanged();
    }
   }
   private DateTime _publicationDate;
   public DateTime PublicationDate
   {
    get { return _publicationDate; }
    set { _publicationDate = value; OnPropertyChanged(); }
   }
   private string _isbn;
   public string ISBN
   {
    get { return _isbn; }
    set { _isbn = value; OnPropertyChanged(); }
   }
  }
 }
```

Now that you have a data model, it is time to write the viewmodel and learn how to communicate with the Web API service.

## Creating the ViewModel

The viewmodel will be responsible for exposing data to the user interface and commands that execute actions over data. You can call the viewmodel class **`BookViewModel`** and make it derive from **`NotifyBase`**. The class needs to expose a collection of **`Book`** objects and an individual **`Book`** instance that will be later bound to the selected item in a **`CollectionView`** in the user interface. The definition can start as follows:

```
using BookClient.Model;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Xamarin.Forms;
namespace BookClient.ViewModel
{
 public class BookViewModel : NotifyBase
 {
  private ObservableCollection<Book> _books;
  public ObservableCollection<Book> Books
  {
   get { return _books; }
   set { _books = value; OnPropertyChanged(); }
  }
  private Book _selectedBook;
  public Book SelectedBook
  {
   get { return _selectedBook; }
   set { _selectedBook = value; OnPropertyChanged(); }
  }
  private const string baseAddress = “https://yourapiservice.
  azurewebsites.net/api/books”;
  public BookViewModel()
  {
   Books = new ObservableCollection<Book>();
  }
```

Note how the constructor creates an instance of the **`Books`** collection in order to avoid null references. Additionally, note how a constant called **`baseAddress`** is defined and contains the address of your Web API service. The value of this constant must be replaced with the URL you previously retrieved in the Azure Management Portal or, if you are using an on-premise server, the address of the Web API service on your server machine. The next step is to define commands that add, edit, and delete data. Data can be added as follows:

```
public Command AddCommand
{
 get
 {
  return new Command(async () =>
  {
   var newBook = new Book();
   newBook.ID = Books.Max(b => b.ID) + 1;
   Books.Add(newBook);
   await PostBookAsync(newBook);
  });
 }
}
```

This command creates an instance of the **`Book`** class and automatically assigns its **`ID`** property with a value that is higher than the highest existing ID value in the collection, retrieved by the Max extension method from the **`IEnumerable`** interface. This is done to provide a valid ID without having the user to do this manually. The command finally invokes a method called **`PostBookAsync`**, which will be defined in the upcoming sections, so ignore any error squiggles in the code editor. The command for deleting an object has simpler implementation:

```
public Command DeleteCommand
{
 get
 {
  return new Command(async () =>
  {
   Books.Remove(SelectedBook);
   await DeleteBookAsync(SelectedBook);
  });
 }
}
```

In short, the object that is currently selected in the user interface is removed from the collection first and then removed remotely via a method called **`DeleteBookAsync`**, which will also be described in the upcoming sections.

Updating an existing **Book** instance is even simpler and is accomplished via the following command definition:

```
public Command UpdateCommand
{
 get
 {
  return new Command(async () =>
  {
   await PutBookAsync(SelectedBook);
  });
 }
}
```

A method called **PutBookAsync** will send the instance of the currently selected object to the Web API, but nothing else needs to be done because edits are already in the **Books** collection. At this point, you need to define all the aforementioned methods that allow for interacting with a Web API service.

## Calling Web API services from C#

In the .NET development, there are many options to work with remote services, but the **HttpClient** class from the **System.Net.Http** namespace is the most common option, especially with Web API services and certainly in Xamarin.Forms. The **HttpClient** class easily allows for working against REST-based services, like Web API, by exposing, among other things, methods that map the HTTP verbs, such as **GetAsync**, **PostAsync**, **PutAsync**, and **DeleteAsync**. Its **BaseAddress** property, of type **Uri**, can be used to store the root address of the web service. While not necessary in the current example, it also exposes a collection called **DefaultRequestHeaders** that you can populate with objects of type **MediaTypeWithQualityHeaderValue**, each representing a request header in the form of a key/value pair. If look back at *Figure 12.9*, where you saw request headers in Postman, you can better understand the purpose of this collection. For the current example, you will create instances of the **HttpClient** where appropriate, set its **BaseAddress** property, and invoke the aforementioned asynchronous methods. For more information, you can refer to the official documentation at **https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient**.

The first operation you want to implement is to retrieve a list of books from the Web API. To do this, write the following method (comments will follow):

```
public async Task GetBooksAsync()
```

```
{
 using (var client = new HttpClient())
 {
  client.BaseAddress = new Uri(baseAddress);
  var result = await client.GetAsync(client.BaseAddress);
  if (result.IsSuccessStatusCode)
  {
   var resultContent = await result.Content.
   ReadAsStringAsync();
   var deserializedBooks =
    JsonConvert. DeserializeObject<List<Book>>
    (resultContent);
   foreach (var book in deserializedBooks)
   {
    Books.Add(book);
   }
  }
 }
}
```

Unless you need to share the same instance of the **`HttpClient`** class across multiple methods, it is recommended to create one with a using block, which ensures that it is disposed after usage. As you can see, the **`BaseAddress`** property is assigned with the root address of the web service, assigned to the **`baseAddress`** field. No address customizations are needed in this case, but you will see an example when deleting data. The next step is calling the **`GetAsync`** method pointing to the service address. This method returns an object of type **`HttpResponseMessage`**, which not only contains the data but also other information sent from the service. For example, the **`IsSuccessStatusCode`** property returns true if the call was successful, whereas the **`StatusCode`** property returns a value from the **`HttpStatusCode`** enumeration that represents the HTTP status code sent by the service; for example, a **`NotFound`** represents the **`404 error`**, or **`BadRequest`** represents the **`400 error`**, whereas **`Success`** represents the **`200 status`** code. In the preceding code, if a response is sent with success, the value of the **`Content`** property of the **`HttpResponseMessage`** class is read in the form of a string via the **`ReadAsStringAsync`**. Content includes the part of the response that is strictly related to data, and **`ReadAsStringAsync`** retrieves the JSON of the data. Based on the current sample data returned by the web service, the result of **`ReadAsStringAsync`** is the following JSON:

```
[{“id”:1,”title”:”Practitioner’s Guide to Data
Science”,”publicationDate”:”2022-01-
01T00:00:00”,”isbn”:”9789391392871”},{“id”:2,”title”:”IoT for
Beginners”,”publicationDate”:”2021-12-
01T00:00:00”,”isbn”:”9789355510068”},{“id”:3,”title”:”iOS 15
Application Development for Beginners”,”publicationDate”:”2021-
12-01T00:00:00”,”isbn”:”9789355511102”}]
```

Obviously, you cannot use JSON directly in C#, so this is where the **`Newtonsoft.Json`** library comes in. The **`JsonConvert`** class exposes a static method that allows converting JSON to C# objects (known as **deserialization**) and converting C# objects to JSON (known as **serialization**). The response from the Web API is a JSON array of **`Book`** objects, so the **`JsonConvert.DeserializeObject`** method converts the string into a **`List<Book>`** instance. You should not convert to an **`ObservableCollection`** directly because **`DeserializeObject`** is intended to work with primitive types only. In fact, the last part of the method iterates over the retrieved **`List`** and adds each **`Book`** object to the **`Books`** collection. Writing objects works a bit differently from retrieving data, but the approach is the same for both **`POST`** and **`PUT`** requests. Consider the following **`PostBookAsync`** and **`PutBookAsync`** methods:

```
public async Task PostBookAsync(Book data)
{
 using (var client = new HttpClient())
 {
  client.BaseAddress = new Uri(baseAddress);
  string json = JsonConvert.SerializeObject(data);
  var content = new StringContent(json, Encoding.UTF8,
      “application/json”);
  var result = await client.PostAsync(client.BaseAddress,
  content);
 }
}
public async Task PutBookAsync(Book data)
{
 using (var client = new HttpClient())
 {
  client.BaseAddress = new Uri(baseAddress);
  string json = JsonConvert.SerializeObject(data);
  var content = new StringContent(json, Encoding.UTF8,
      “application/json”);
  var result = await client.PutAsync(client.BaseAddress,
  content);
 }
}
```

In both, after creating an instance of the **`HttpClient`** class, you first need to convert to a JSON string the instance of the Book you want to send to the

Web API. This is accomplished by invoking the **`JsonConvert.SerializeObject`** method, which receives the object you want to convert as the parameter. Both **`PostAsync`** and **`PutAsync`** need to send JSON content that is adapted to HTTP communications. For this purpose, you can use the **`StringContent`** class, passing the JSON data, the encoding format (UTF8 is the appropriate encoding), and the **`application/json`** specification to the constructor. The resulting object is passed to both **`PostAsync`** and **`PutAsync`**, along with the service address. The last method is called **`DeleteBookAsync`**. It first removes the specified **`Book`** instance from the collection and then invokes **`DeleteAsync`** on the **`HttpClient`** instance to send a **`DELETE`** request to the Web API. The way you set up the request is like for the previous two methods, but the difference is in the target address. In fact, the Web API requires clients to append the ID of the object to the service address. You can do this in many ways, but in the current example, the ID is appended to the value of the **`BaseAddress`** property:

```
   public async Task DeleteBookAsync(Book data)
   {
    Books.Remove(data);
    using (var client = new HttpClient())
    {
     client.BaseAddress = new Uri(baseAddress);
     string json = JsonConvert.SerializeObject(data);
     var content = new StringContent(json, Encoding.UTF8,
         “application/json”);
     var result = await client.
      DeleteAsync($”{client.BaseAddress}/{data.ID}”);
    }
   }
  }
 }
```

The viewmodel is now complete, and you are ready to write a minimal user interface that allows for displaying and editing data.

**As an exercise, you could implement error handling with `try..catch` blocks and make methods return a value depending on the status code sent by the Web API. You could also move the `GetBooksAsync`, `PostBookAsync`, `PutBookAsync` and `DeleteBookAsync`, methods to a separate service class.**

## Designing the user interface

The purpose of the user interface is to display a list of books inside a **`CollectionView`** view and provide buttons to add, edit, and remove data. All the necessary XAML markup can be written inside the **`MainPage.xaml`** file. In addition, all the XAML markup is based on techniques you learned in *Chapter 9, Resources and Data Binding*, so they will not be explained again here. The following is the XAML code for the sample user interface:

```
<?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="BookClient.MainPage">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <CollectionView x:Name="BookList" SelectionMode="Single"
        SelectedItem="{Binding SelectedBook}">
      <CollectionView.ItemTemplate>
        <DataTemplate>
          <StackLayout Orientation="Vertical">
            <Entry Margin="10,10,10,0" Text="{Binding Title}"
            FontSize="16" FontAttributes="Bold"/>
            <Entry Margin="10,0,10,0" Text="{Binding ISBN}"
            FontSize="12"
            FontAttributes="Italic"/>
            <DatePicker Margin="10,0,10,0" Date="{Binding
            PublicationDate}" FontSize="14" />
          </StackLayout>
        </DataTemplate>
      </CollectionView.ItemTemplate>
    </CollectionView>
    <StackLayout Grid.Row="1" Orientation="Horizontal">
      <Button x:Name="AddButton" Text="Add new" Command="{Binding
      AddCommand}"
      Margin="10,0,0,0"/>
      <Button x:Name="DeleteButton" Text="Delete"
      Command="{Binding DeleteCommand}" Margin="5,0,0,0" />
      <Button x:Name="UpdateButton" Text="Update"
      Command="{Binding UpdateCommand}" Margin="5,0,0,0"/>
    </StackLayout>
  </Grid>
</ContentPage>
```

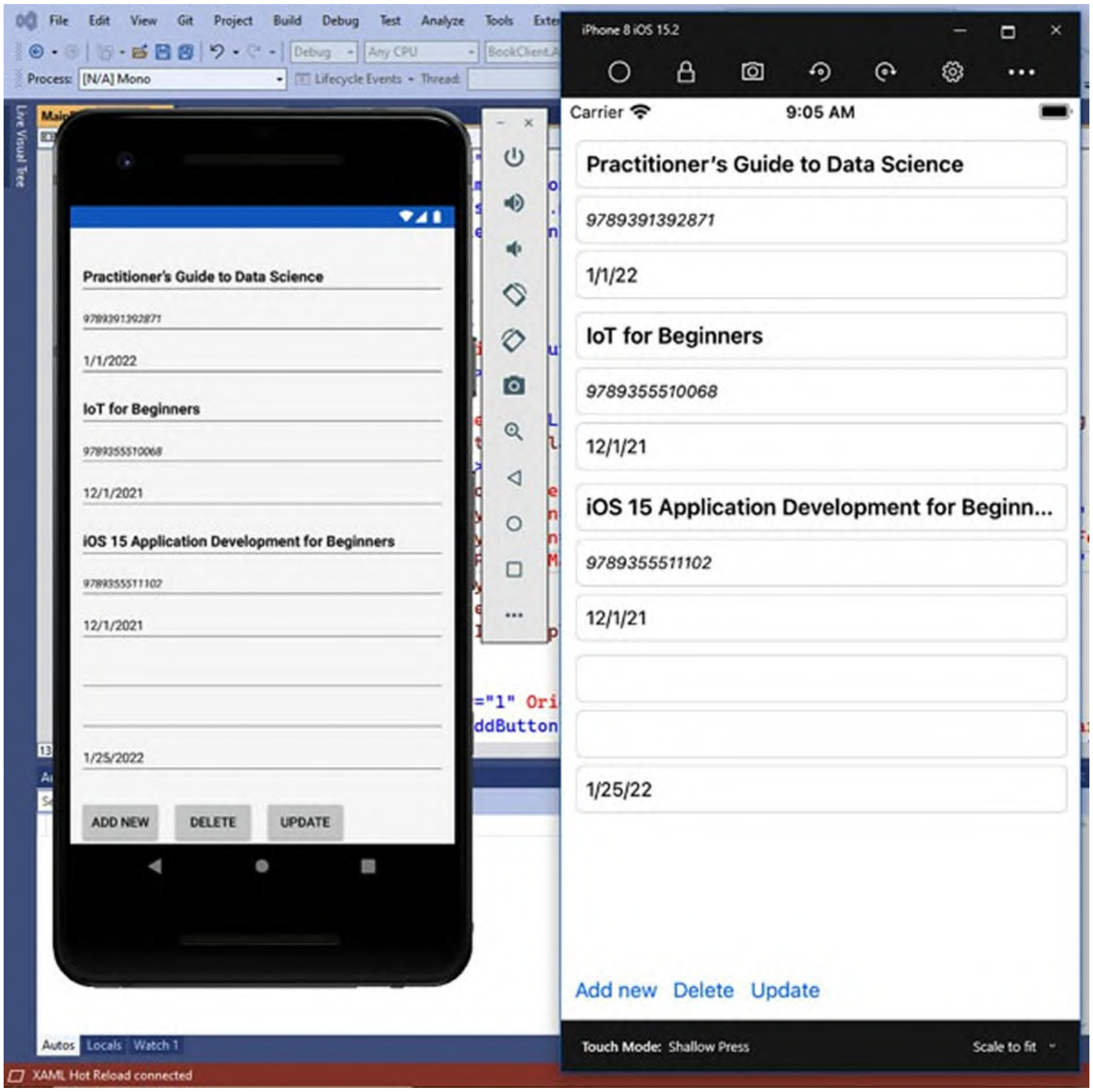
The very last part of the sample project is to create an instance of the viewmodel and assign it to the binding context of the page, as follows:

```
using BookClient.ViewModel;
using Xamarin.Forms;
namespace BookClient
{
  public partial class MainPage : ContentPage
  {
    private BookViewModel ViewModel { get; set; }
    public MainPage()
    {
      InitializeComponent();
      ViewModel = new BookViewModel();
      BindingContext = ViewModel;
      BookList.ItemsSource = ViewModel.Books;
    }
    protected override async void OnAppearing()
    {
      base.OnAppearing();
      await ViewModel.GetBooksAsync();
    }
  }
}
```

Note that the **`GetBooksAsync`** method from the viewmodel is invoked in the **`OnAppearing`** method because this allows for asynchronous method calls. Now, you have everything you need to get the result of this long work in both Android and iOS devices.

## Testing the application

If you run the client application, you will see the list of books returned by the Web API, as shown in *Figure 12.21*, where you can also see an empty record added by clicking the **`Add new`** button. This button causes the bound command to insert a new object into the data store, so if you make changes to it, you will click the **`Update`** button:



Practitioner's Guide to Data Science
9789391392871
1/1/2022
IoT for Beginners
9789355510068
12/1/2021
iOS 15 Application Development for Beginners
9789355511102
12/1/2021

1/25/2022

ADD NEW DELETE UPDATE

Practitioner's Guide to Data Science
9789391392871
1/1/22
IoT for Beginners
9789355510068
12/1/21
iOS 15 Application Development for Beginn...
9789355511102
12/1/21

1/25/22

Add new Delete Update

***Figure 12.21:*** *Consuming a Web API from mobile apps*

If you want to test the **`Delete`** button, select a book in the list first.

## Conclusion

It is very common for mobile apps to communicate with remote services to exchange data and information. Applications you build with Xamarin.Forms can interact with any REST-based services, but the most common way to build services with the Microsoft stack is by working with Web API. This chapter explained the basics of the different data exchange formats, focusing on JSON, and then explored how to create and publish a Web API service that exposes calls that can be invoked to perform operations against data. In the second part of the chapter, you learned how to call Web API from a Xamarin.Forms project and how to connect data to the user interface reusing existing skills. In the next chapter, you will work with native platform API for a full programming experience.

## Key terms

- **REST**: The acronym for REpresentational State Transfer, an architectural style that drives development of web software through constraints.
- **JSON**: The acronym for JavaScript Object Notation, a standard format used to exchange data between services and client applications.
- **Web API**: A web service that exposes methods that can be invoked to perform operations over a network via HTTP verbs.
- **HttpClient**: A class that makes it possible to invoke Web API calls in .NET.

## Suggested readings

- Microsoft ASP.NET Web APIs documentation (**https://dotnet.microsoft.com/en-us/apps/aspnet/apis**).
- Publishing Web API services to Azure (**https://docs.microsoft.com/en-us/aspnet/core/tutorials/publish-to-azure-api-management-using-vs**).
- Publishing Web apps to Internet Information Services (**https://docs.microsoft.com/en-us/aspnet/core/tutorials/publish-to-iis**).
- Consuming REST services from Xamarin.Forms (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/data-cloud/web-services/rest**).
- Using Web API with Entity Framework (**https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/**).

# CHAPTER 13

# Working with Native API

## Introduction

One of the first concepts you have learned about Xamarin.Forms is that, as a cross-platform technology, it provides a common way to access features that are available on all the supported systems. However, sometimes you might need to make a step forward and leverage specific features that require accessing native APIs via Xamarin.iOS and Xamarin.Android code. As an example, that is very common in the real-world development, removing the border of an Entry view requires accessing the native API of the view itself on all the target platforms. In this chapter, you will learn how to access native APIs and features of the iOS and Android systems from C# and understand how you can call native features with a cross-platform approach using the Xamarin.Essentials library. As you can imagine, it is not possible to discuss the entire native API and codebase of iOS and Android in one chapter, so you will be put in the conditions to extend your knowledge on your own after learning the crucial concepts. This is also the last technical chapter of the book, which will complete the foundations of mobile app development with Xamarin.Forms.

## Structure

In this chapter, we will cover the following topics:

- Working with the device class
- Advanced view customizations: Custom renderers
- Managing native properties with effects
- Displaying native views
- Customizing views with platform-specifics
- Cross-platform access to native API: Xamarin.Essentials

## Objectives

After completing this chapter, you will be able to run code blocks only on specific devices, and you will know how to change the structure and behavior of views. In addition, this chapter will help you learn how to include native iOS and Android views inside Xamarin.Forms code and how to access native features without using native APIs, with the help of the Xamarin.Essentials library.

## Preparing a sample project

This chapter comes with two companion solutions that you can open with Visual Studio to better follow the examples. One is called `NativeAccess` and relates to accessing device and system features, and one is called XamarinEssentials, which is about the usage of the Xamarin.Essentials library.

If you wish to create new project on your own, follow these steps:

1. Create two new Xamarin.Forms solutions called `NativeAccess` and `XamarinEssentials`, respectively, for consistency with the sample solution.
2. Do not edit or remove the auto generated `MainPage.xaml` file; it will be used later.
3. For each new feature discussed in the chapter, add a new item of type `Content Page (XAML)`. To accomplish this, right-click on the shared project name and then click on `Add New Item` in the `Context` menu.
4. In the `Add New Item` dialog, click on the Xamarin.Forms node on the left and then select the `Content Page (XAML)` item template.
5. Assign to the new XAML file a name that matches the discussed topic and click on `Add`.

For each page you add to the project, add an empty `StackLayout` to the `ContentPage` and assign its `VerticalOptions` property with `CenterAndExpand`. Unless specified, this will be the layout of choice for the code examples in the upcoming pages.

## Working with the device class

Sometimes, you might need to run some code only on a specific system or on a specific device, such as phone, tablet, or desktop computer. You can do this through a class called **`Device`**. Among other things, it exposes two properties called **`RuntimePlatform`** and **`Idiom`**, where the first property returns the system, the application is running on, and the second one returns the type of device. For better understanding, consider the following code:

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

**`RuntimePlatform`** is of type **`string`** and returns a value that represents the current system, among iOS, Android, UWP (Windows 10 and higher), macOS, and WPF. The **`GetNamedSize`** method returns the named size of the current font from the target system and is another way to retrieve device-specific information. For system colors, you can instead call the **`GetNamedColor`** method. Another common use of **`RuntimePlatform`** is inside a simple if block, like the following:

```
if(Device.RuntimePlatform == Device.Android)
{
// Executes the code here only on Android devices
}
```

Sometimes, it is also useful to understand the form factor of the current device, which can be accomplished as follows:

```
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;
}
```

The **`Idiom`** property, of type **`TargetIdiom`**, allows you to understand the current type of device. It can be useful if you need to make UI adjustments based on the device type, and it can be combined with **`RuntimePlatform`**.

## Working with Timers

In Xamarin.Forms, the **`Device`** class is also used to create timers. A timer is an object that allows executing code only at the specified interval of time. This is accomplished with the **`StartTimer`** method, which takes an object of type **`TimeSpan`** and the action to be executed as parameters. The following code shows an example:

```
public MainPage()
{
 InitializeComponent();
 Device.StartTimer(TimeSpan.FromSeconds(30), ShowMessage);
}
private bool ShowMessage()
{
 DisplayAlert(“Info”, “30 seconds have passed”, “OK”);
 return true;
}
```

In the example, the **`ShowMessage`** method is invoked every 30 seconds. When you want to stop the timer, your method should return **`false`**; otherwise, it should return **`true`** until you want the timer to run.

## Running thread-safe code

With advanced programming techniques, you might face situations where code started from the user interface thread will work on a separate new thread. If the code running on a separate thread needs to interact with the elements of the user interface, the runtime will throw an exception for cross-thread violation. In such situations, you can invoke the **`Device.BeginInvokeOnMainThread`** method, which takes the action to be executed on the UI thread as parameter. This is an example:

```
Device.BeginInvokeOnMainThread(ActionToExecute);
…
private void ActionToExecute()
{
  // This runs on the UI thread
}
```

There is also an asynchronous alternative called **`InvokeOnMainThreadAsync`**, which you can use with the await operator, as follows:

```
await Device.InvokeOnMainThreadAsync(ActionToExecute);
…
private async Task ActionToExecute()
{
  // This runs on the UI thread
}
```

**Many developers realize that they need to use the aforementioned methods only when the debugger throws a thread violation exception, which is quite normal when you are focused on coding. Especially at the beginning of your career, this is likely to be your case too. Do not worry; always debug with attention, and you will be able to fix your code accordingly.**

## Device-based content orientation

If you plan to build apps that will be distributed globally, you need to support different content orientation. In fact, there are cultures that require orienting content from left to right and cultures that require orienting content from right to left. The **`Device`** class helps you accomplish this directly in XAML by assigning the **`FlowDirection`** property as follows:

```
<ContentPage FlowDirection=”{x:Static Device.FlowDirection}”>
```

Generally speaking, the **`FlowDirection`** property (available to each view) allows for specifying content orientation in order to support different cultures. Pointing to the static, same-named property of the **Device** class (**x:Static** allows for calling static properties from XAML), you will be able to support orientation based on the device settings.

## Conditional XAML: OnPlatform and OnIdiom

Every object is deriving from the **VisualElement** class, so all views and layouts allow for specifying different behaviors according to the target system or device idiom. For example, in most examples so far, you have added a Padding with value 0,20,0,0 to the pages, and this applies to both Android and iOS (and it would also apply to other supported platforms like UWP). However, specifying a padding is only relevant to iOS. For this reason, you can leverage a markup extension called **OnPlatform**, which allows for setting different property values depending on the target system. In the following example, different padding is applied to different systems:

```
<ContentPage.Padding>
  <OnPlatform x:TypeArguments=”Thickness”
     iOS=”0, 20, 0, 0”
     Android=”0, 5, 0, 0”
     UWP=”0, 10, 0, 0” />
</ContentPage.Padding>
```

The following is a list of the key points:

- The property you wish to assign with a value that is different on each platform needs to be declared using the extended syntax.
- You must specify the **OnPlatform** markup extension and supply the data type for the current property via the **x:TypeArguments** tag. In this example, Padding is of type **Thickness**, so the latter is what you need to specify.
- You need to specify one or more supported platforms, like iOS, Android, and UWP (the latter is shown in the code only for educational purposes).
- **OnPlatform** also has an expanded syntax that you can leverage if you want to assign the same value to multiple platforms, like this:

```
<ContentPage.Padding>
  <OnPlatform x:TypeArguments=”Thickness”>
    <On Platform=”iOS” Value=”0,20,0,0”/>
    <On Platform=”Android, UWP” Value=”0,5,0,0”/>
  </OnPlatform>
```

```
    </ContentPage.Padding>
```

**`OnPlatform`** is commonly used to tweak visual elements that have different alignment and stretching options on different platforms. You can also assign different property values depending on the device idiom using the **`OnIdiom`** markup extension. The following example shows how to set the orientation of a **`StackLayout`** depending on the device type:

```
<StackLayout.Orientation>
  <OnIdiom x:TypeArguments=”StackOrientation”>
    <OnIdiom.Tablet Value=”Horizontal” />
    <OnIdiom.Phone Value=”Vertical” />
  </OnIdiom>
</StackLayout.Orientation>
```

You still need to set **`x:TypeArguments`** with the supported data type, and then you set one or more idioms via **`OnIdiom.Tablet`**, **`OnIdiom.Phone`**, and **`OnIdiom.Desktop`**.

## Advanced view customization: Custom renderers

As you know, Xamarin.Forms provides visual elements that are available on all the supported platforms, and for such visual elements, it implements properties and characteristics that are available on all platforms. However, there are situations where you might need access properties that are only available in the native API of a view. There can be plenty of reasons to do this, and most of them are related to the design and implementation of the user experience. For example, suppose you need to implement an extremely common requirement: an Entry that has no border. There is no built-in property that allows you to do this directly, so the only way to accomplish this is by controlling the backing native view via the so-called custom renderers. A renderer is a C# class that wraps a native view into a .NET object. For instance, Xamarin.Forms provides the Label to display static text. Behind the scenes, a class called **`LabelRenderer`** translates the Android’s **`TextView`** and the iOS’ **`UILabel`** into the **`Label`** you know. For each visual element, there is a renderer in all the supported platforms, which means that both iOS and Android implement a **`LabelRenderer`** class that translates into a unified, shared view (the **`Label`**), the native views. The built-in renderers are developed in a way that they expose to the **`Xamarin.Forms`** codebase all the objects that are available cross-platform. This is why your own renderers are referred to as custom renderers. In the next section, you will learn how to implement a custom renderer that removes the border from an **`Entry`** view.

## Defining a custom view

Custom renderers are globally applied to the type they target, so it is always a good idea to define a new view that derives from the one you want to modify before creating a new renderer. In the shared project, add a new class called **`BorderlessEntry`** with the following code:

```
using Xamarin.Forms;
namespace NativeAccess
{
  public class BorderlessEntry: Entry
  {
  }
}
```

This class is a copy of the **`Entry`** view and will be backed by a custom renderer. This makes it possible to maintain regular **`Entry`** views while using custom ones. You will retake this class after defining the custom renderers.

## Defining the Android renderer

In **`Solution Explorer`**, right-click on the Android project and then select **`Add`** | **`Class`**. Add a new class called **`BorderlessEntryRenderer`**.

**Tip: You are free to choose the names for your custom renderer classes; the convention is to combine the name of the custom shared view with the `Renderer` suffix. This will help you with code maintenance and will make it simpler for other developers to understand how renderers are implemented.**

Write the following code, which will be commented below:

```
using Android.Content;
using Android.Graphics.Drawables;
using Android.OS;
using NativeAccess;
using NativeAccess.Droid.Renderers;
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;
[assembly: ExportRenderer(typeof(BorderlessEntry),
typeof(BorderlessEntryRenderer))]
namespace NativeAccess.Droid.Renderers
```

```
{
 public class BorderlessEntryRenderer : EntryRenderer
 {
  public BorderlessEntryRenderer(Context context) :
  base(context)
  {
  }
  protected override void
  OnElementChanged(ElementChangedEventArgs<Entry> e)
  {
   base.OnElementChanged(e);
   if (Control != null)
   {
    Control.Background =
      new ColorDrawable(Android.Graphics.Color. Transparent);
   }
  }
 }
}
```

Every custom renderer overrides the **`OnElementChanged`** method, which is the place where you get the instance of the native view, represented by the **`Control`** property. In this case, the native view is the Android's **`FormTextView`**. Once you have the instance of the native view, you can customize its properties and behavior as required. In this case, the entire background of the view is assigned with a transparent color. It is always important to enclose the code inside a null check (**`if(Control != null)`**) or inside a **`try..catch`** block because a renderer might be invoked before the view is drawn on screen, especially at the application startup. The next relevant point in the code is the **`ExportRenderer`** attribute. It is responsible for telling Xamarin.Forms that, across the entire app (assembly specification), views of type **`BorderlessEntry`** must be rendered using this custom renderer instead of the built-in system renderer (which would be the **`EntryRenderer`** in this case). Now, you can better understand why you created a new Xamarin.Forms view: if you did not, your renderer would have been globally applied to all the Entry views in your app, which is not ideal. The next step is defining a custom renderer for iOS.

## Defining the iOS renderer

Following the same steps, add a new class called **`BorderlessEntryRenderer`** to the iOS project in the solution. This time, the code is as follows:

```
using NativeAccess;
using NativeAccess.iOS;
using System.ComponentModel;
using UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;
[assembly: ExportRenderer(typeof(BorderlessEntry),
typeof(BorderlessEntryRenderer))]
namespace NativeAccess.iOS
{
 public class BorderlessEntryRenderer : EntryRenderer
 {
  protected override void
  OnElementChanged(ElementChangedEventArgs<Entry> e)
  {
   base.OnElementChanged(e);
   if (Control != null)
   {
    // Clips sublayers to the bounds of the root layer of the
    view
    Control.Layer.MasksToBounds = true;
    // No corner radius
    Control.Layer.CornerRadius = 0;
    // No border
    Control.BorderStyle = UITextBorderStyle.None;
    // Minimum border width
    Control.Layer.BorderWidth = 1;
    // Assign transparent colours
    Control.Layer.BorderColor = Color.Transparent.ToCGColor();
   }
  }
 }
}
```

On iOS, you still apply the **`ExportRenderer`** attribute, and you still override the **`OnElementChanged`** method, but this time, the control is an iOS' **`UITextField`**, with its own methods and properties. Now that you have your custom renderers, you can test them in your user interface.

## Applying custom renderers to views

Now that you have defined custom renderers, you need to explicitly use them. In a page of your shared project, you first need to declare an XML namespace that references the project and a namespace where the target view is defined. If you follow the companion solution, the **`BorderlessEntry`** class is defined in the root namespace, which you reference as follows:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:NativeAccess"
    x:Class="NativeAccess.CustomRenderersPage">
```

Now, you can use the view as you would use any other view, declaring it as follows:

```
<ContentPage.Content>
  <StackLayout Orientation="Vertical" Padding="20">
   <Label Text="Regular Entry:" />
   <Entry WidthRequest="150" BackgroundColor="Yellow"/>
   <Label Text="Borderless Entry:"/>
   <local:BorderlessEntry WidthRequest="150"
   BackgroundColor="Yellow" />
  HorizontalOptions="FillAndExpand"/>-->
  </StackLayout>
</ContentPage.Content>
```

If you run this code, the runtime will apply the appropriate custom renderer depending on the platform the app is running on. *Figure 13.1* shows the result of the example, with a regular **`Entry`** and a **`BorderlessEntry`**:

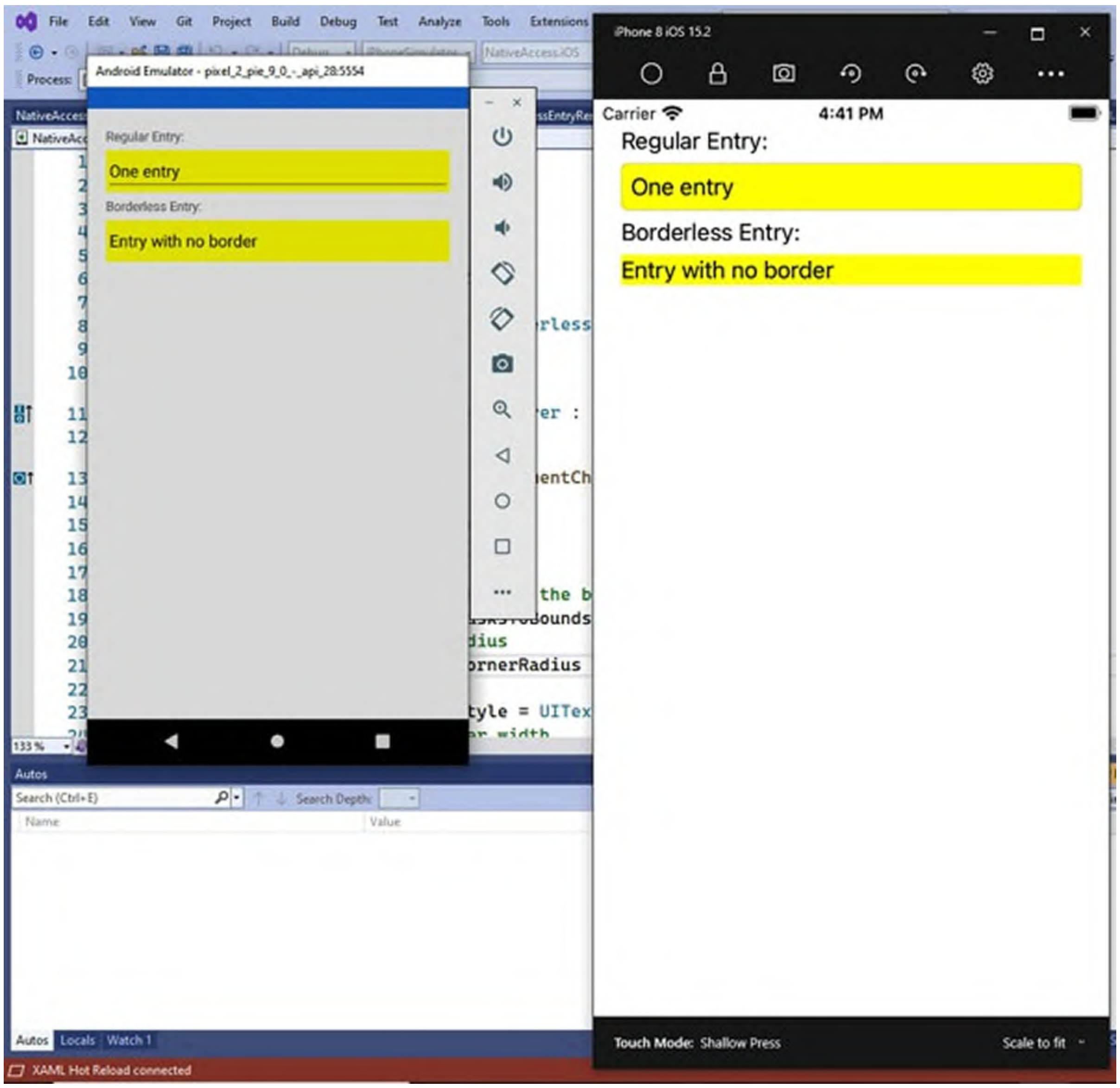


***Figure 13.1:*** *Applying a custom renderer*

## More information on custom renderers

Because every visual element in Xamarin.Forms relies on renderers and on native Android and iOS views, and creating custom renderers totally depends on your requirements, it is not possible to discuss custom renderers further, but it is convenient to summarize a few key points that will be useful for you as a developer:

- It is best practice to define a dedicated view in the shared project.
- Custom renderers for all platforms are applied via the **`ExportRenderer`**

attribute.

- In most cases, you override the **`OnElementChanged`** method to get the instance of the native view on all supported platforms.
- You can create custom renderers even for third-party components.
- Referring to the Xamarin.iOS (**https://docs.microsoft.com/en-us/xamarin/ios/user-interface/controls/**) and Xamarin.Android (**https://docs.microsoft.com/en-us/xamarin/android/user-interface/controls/**) native views documentation is the most important thing to do when you need to address a requirement that has no cross-platform implementation.

Having that said, it is also important to bookmark the official documentation page about custom renderers, which is **https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/**.

## Managing native properties with effects

Sometimes, you might want to implement customizations on native views, but you only need to change some of their properties, without redefining the overall behavior of the view itself. In this case, you can avoid custom renderers and take advantage of effects. An effect is a class that derives from **`Xamarin.Forms.RoutingEffect`** in the shared code, with platform-specific implementations that derive from **`Xamarin.Forms.Platform.Android.PlatformEffect`** and **`Xamarin.Forms.Platform.iOS.PlatformEffect`**, respectively. For better understanding, suppose you want to specify a maximum length for the text inside an **`Entry`**. At this purpose, consider the following XAML code:

```
<Label Text=”Entry with an effect:” />
<Entry WidthRequest=”120” x:Name=”DataEtry”>
  <Entry.Effects>
    <local:MaxLengthEffect MaxLength=”8” />
  </Entry.Effects>
</Entry>
```

**Tip: For convenience, the preceding code can be added after the two `Label`/`Entry` groups of the example about custom renderers.**

Every view exposes a collection called **`Effects`** and can contain one or more effects. In this example, an effect called **`MaxLengthEffect`** is applied, and a property called **MaxLength** is assigned with an integer. Now, the goal is to implement the effect and the platform-specific code for both Android and iOS.

## Declaring effects

In your shared project, add a new class called **MaxLengthEffect** with the following code:

```
public class MaxLengthEffect : RoutingEffect
{
  public MaxLengthEffect() : base($”MyCompanyName.
  {nameof(MaxLengthEffect)}”)
  {
  }
  public int MaxLength { get; set; }
}
```

The class inherits from **RoutingEffect** so that **Xamarin.Forms** knows that it is an effect. The constructor contains a reference to a so-called resource group. A resource group can be considered as a container of effects, so you define a root prefix (**MyCompanyName** in this case) followed by a dot, and then you supply an identifier for the effect. By convention, you use the name of the effect itself. Defining a resource group and an effect identifier is important because it is used by **Xamarin.Forms** to identify and apply an effect. Then, you can define all the properties and methods your effect needs. For the current example, you only need an int property that can be assigned in XAML, as you saw at the beginning of this section. There is nothing else to do on the side of the shared code unless you want to customize the effect further.

## Implementing platform-specific effects

Like custom renderers, effects also have a platform-specific implementation, which is necessary to set the native view’s properties. Starting with the Android implementation, add a new class called **MaxLengthEffectPlatform** to the **Droid** project with the following code:

```
using Android.Widget;
using NativeAccess;
using NativeAccess.Droid;
using System;
using System.Linq;
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;
[assembly: ResolutionGroupName("MyCompanyName")]
[assembly: ExportEffect(typeof(MaxLengthEffectPlatform),
nameof(MaxLengthEffect))]
namespace NativeAccess.Droid
{
 public class MaxLengthEffectPlatform : PlatformEffect
 {
  protected override void OnAttached()
  {
   try
   {
    var sharedEffect = (MaxLengthEffect)
     Element.Effects.FirstOrDefault(e => e is
     MaxLengthEffect);
    TextView editEntry = Control as TextView;
    editEntry?.SetFilters(new Android.Text.IInputFilter[]
    {
     new Android.Text.
     InputFilterLengthFilter(sharedEffect.MaxLength)
    });
   }
   catch (Exception ex)
   {
    //Catch any exception
   }
  }
  protected override void OnDetached()
  {
  }
 }
}
```

The namespace block must be decorated with the **`ResolutionGroupName`** attribute, which specifies the resolution group that the current effect belongs to. Similarly, to custom renderers, where you apply the **`ExportRenderer`** attribute, you also need to apply the **`ExportEffect`** attribute, passing the name of the platform-specific implementation and the shared effect name as parameters. Deriving from the **`PlatformEffect`** class requires implementing two methods: **`OnAttached`** and **`OnDetached`**. The first method is invoked when the effect is being applied, whereas the second one is invoked when the effect is being removed. **`OnDetached`** is only necessary when you have code that explicitly removes the effect from the collection. What **`OnAttached`** does in the previous code is listed here:

- It retrieves the first effect of type **`MaxLengthEffect`** from the caller view (**`Element`**).
- It retrieves the native Android **`TextView`**, converting **`Control`**, of type object, into a specialized instance.
- It adds a new filter of type **`InputFilterLengthFilter`** to the **`TextView`** via the **`SetFilters`** method.

With this code, it will not be possible to write more than the specified number of characters. This will be demonstrated further on in the chapter, but first you need to add the platform-specific implementation for iOS. Add a **`MaxLengthEffectPlatform`** class to the iOS project and write the following code:

```
using Foundation;
using NativeAccess.iOS;
using System;
using System.Linq;
using UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;
[assembly: ResolutionGroupName(“MyCompanyName”)]
[assembly: ExportEffect(typeof(MaxLengthEffectPlatform),
“MaxLengthEffect”)]
namespace NativeAccess.iOS
{
 public class MaxLengthEffectPlatform : PlatformEffect
 {
  protected override void OnAttached()
  {
   try
   {
    var sharedEffect =
     (MaxLengthEffect)Element.Effects.
     FirstOrDefault(e => e is MaxLengthEffect);
    UITextField nativeEntryView = Control as UITextField;
    if (nativeEntryView != null)
    {
     nativeEntryView.ShouldChangeCharacters = (UITextField
     textField,
      NSRange range, string replacementString) =>
     {
      var length = textField.Text.Length - range.Length +
        replacementString.Length;
      return length <= sharedEffect.MaxLength;
     };
```

```
        }
      }
      catch (Exception ex)
      {
        //Catch any exception
      }
    }
    protected override void OnDetached()
    {
    }
  }
}
```

In iOS, the **`UITextField`** native view does not include the capability to directly add filters, so the code assigns the **`ShouldChangeCharacters`** property with an anonymous method that replaces the original content of the view with a string whose length is limited by the **`MaxLength`** property of the effect.

## Testing the code

If you run the sample application, you will get a result similar to what you see in *Figure 13.2*:

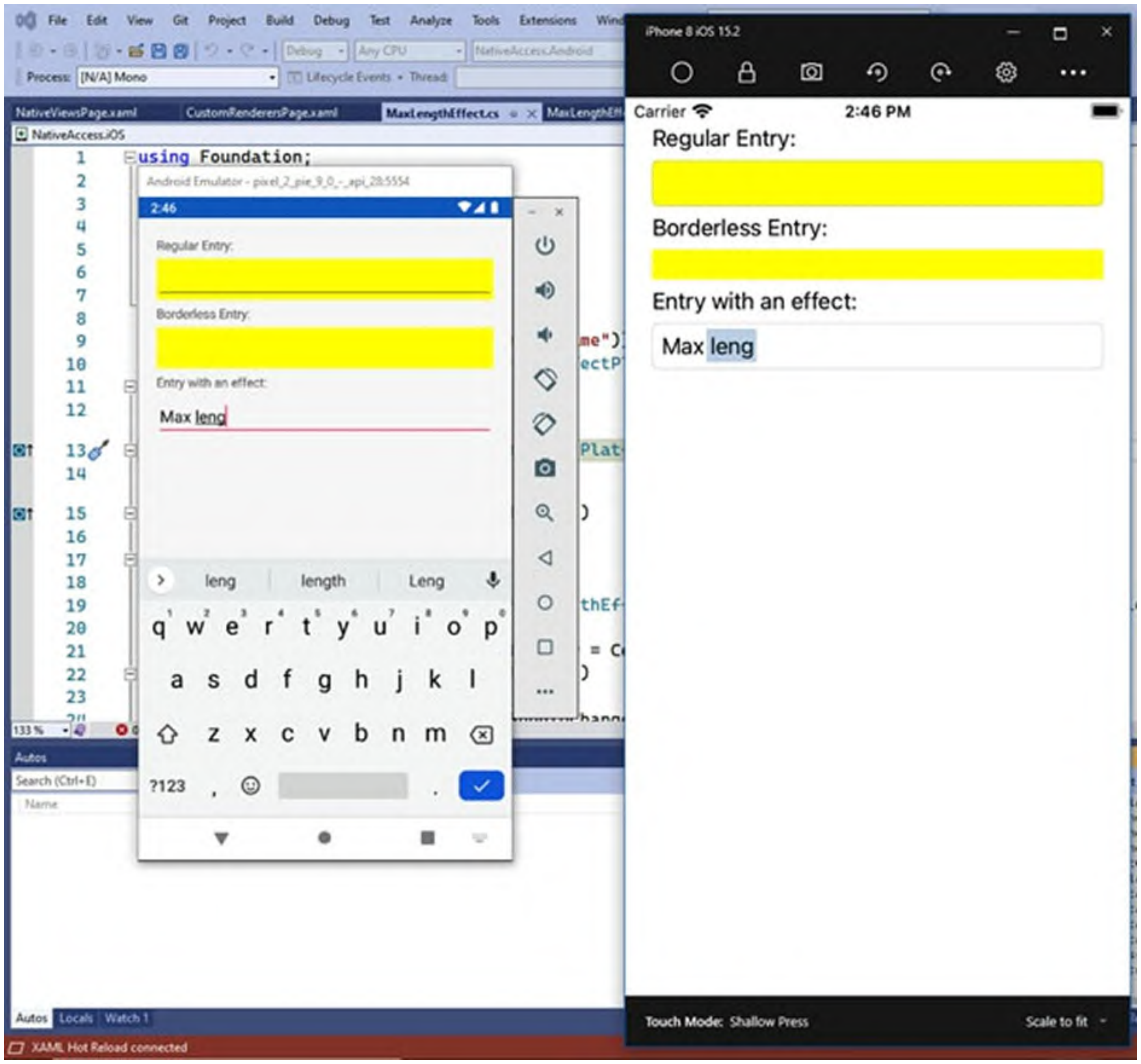


***Figure 13.2:*** *Applying an effect*

If you type inside both Entry views, you will be prevented from adding more than the specified number of characters. Effects can be useful in many situations, but their usage makes sense only when you do not need to redefine the behavior of the native view.

## Displaying native views

Xamarin.Forms also gives you the option to embed native Android and iOS views in your XAML. Assuming that you have a page where you want to test this scenario, you must declare the following XML namespaces:

```
<?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"
    x:Class="NativeAccess.NativeViewsPage">
  <ContentPage.Content>
</ContentPage>
```

For iOS, you need to import the **`UIKit`** namespace from the **`Xamarin.iOS`** library. For Android, you need two namespaces: **`Android.Widget`**, which allows embedding native views, and the partial **`Xamarin.Forms`** namespace from the **`Xamarin.Forms.Platform.Android`** library, which is necessary to retrieve the context of the user interface. Once you have these namespaces, you can quickly embed native views. The following example demonstrates how to show native labels (**`TextView`** for Android and **`UILabel`** for iOS):

```
<StackLayout>
  <ios:UILabel Text="Native Text"
  View.HorizontalOptions="Start"/>
  <androidWidget:TextView Text="Native Text"
       x:Arguments="{x:Static formsandroid:Forms.Context}" />
</StackLayout>
```

As you can see, for Android, you need to pass the UI context to the constructor of the view, with the **`x:Staticformsandroid:Forms.Context`** syntax. You can then assign properties, invoke methods and handle events exactly as you would do with any other **`Xamarin.Forms`** view.

**Tip: Knowing how to embed native views in your shared code is part of your knowledge, but it is something that you might want to avoid when possible. You should only take advantage of this feature when you must include a view for which there is no Xamarin.Forms counterpart. Also, do not forget that the XAML editor has limited support for this feature.**

## Customizing views with platform-specifics

The biggest benefit of **`Xamarin.Forms`** is that it allows consuming, from a shared codebase, views with their properties that are available on all the supported platforms. Previously, you saw how you can take advantage of custom renderers to expose to **`Xamarin.Forms`** features that are not cross-platform. For simpler situations, **`Xamarin.Forms`** provides the so-called **platform-specifics**. This feature simplifies consuming functionalities that are only available on a given platform. This means that a platform-specific available for Android might not be available for iOS and vice versa. For example, the separator in the **`ListView`** on iOS is not full width. Instead of creating a custom renderer, you can leverage a platform specific. This is accomplished with the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:Xamarin.Forms.
    PlatformConfiguration.iOSSpecific;assembly=Xamarin.
    Forms.Core"
    x:Class="NativeAccess.ExamplePlatformSpecifics">
  <ContentPage.Content>
   <StackLayout>
    <ListView ios:ListView.SeparatorStyle="FullWidth"
    x:Name="FullSeparatorListView">
    </ListView>
   </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

On iOS, you need to import the **`Xamarin.Forms.PlatformConfiguration.iOSSpecific`** namespace, whereas on Android, you need to import the **`Xamarin.Forms.PlatformConfiguration.AndroidSpecific`** namespace. In order to consume a platform-specific, you type the namespace identifier followed by a colon and the property you want to customize. The following Android platform-specific instead demonstrates how to enable zooming contents on a **`WebView`**:

```
<?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:android="clr-namespace:Xamarin.Forms.
    PlatformConfiguration.
    AndroidSpecific;assembly=Xamarin.Forms.Core"
    x:Class="NativeAccess.PlatformSpecificsPage">
  <ContentPage.Content>
   <StackLayout>
    <WebView Source="https://www.microsoft.com"
     android:WebView.EnableZoomControls="true"
     android:WebView.DisplayZoomControls="true" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

The code is easy to understand. The real point is that the first platform-specific is only available on iOS, whereas the second one is only available on Android, but they both avoid implementing custom renderers for these simple situations. The official documentation provides a long list of platform-specifics for both Android (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/android/**) and iOS (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/ios/**). You also have the option to apply platform-specifics in C# code. This can be accomplished by invoking the **`On`** extension method, as follows:

```
WebView1.On<Android>
().EnableZoomControls(true).DisplayZoomControls(true);
```

You specify the target platform as the method's type parameter, and then, with the help of IntelliSense, you can walk through the list of available platform-specifics. This is another way to discover what is available if you do not want to read the documentation. Obviously, you first need to add appropriate using directives pointing to the platform-specific namespaces before you can invoke the **`On`** method and its members.

## Accessing device features: The DependencyService class

So far, you have seen how to access native API in order to customize elements of the user interface, but this is only a part of what you can do. In fact, sometimes you will need to access platform-specific features from the device hardware or the operating system, which need to be consumed from the shared project. **`Xamarin.Forms`** provides the **`DependencyService`** class to work with these scenarios. The programming pattern based on this class requires you to define an interface in the shared project, with one or more members that will need a platform-specific implementation and will do the real work. For better understanding, imagine that you need to retrieve the language set on the operating system. Add the following interface to the shared project:

```
public interface IDeviceInfo
{
  string GetSystemLanguage();
}
```

The shared code will invoke a **`GetSystemLanguage`** method to retrieve the system language, and the implementation is demanded to the platform-specific projects. In the Android project, add a new class called **`DeviceInfo`** with the following code:

```
using System.Globalization;
namespace NativeAccess.Droid
{
 public class DeviceInfo : IDeviceInfo
 {
  public string GetSystemLanguage()
  {
   return CultureInfo.CurrentCulture.ToString();
  }
 }
}
```

On Android, you can simply retrieve the current OS language via the **`CultureInfo.CurrentCulture`** object. On iOS, things work differently, which is why you need two different implementations of the interface. Add a **`DeviceInfo`** class to the iOS project with the following code:

```
using Foundation;
namespace NativeAccess.iOS
{
 public class DeviceInfo : IDeviceInfo
 {
  public string GetSystemLanguage()
  {
   return NSLocale.PreferredLanguages[0];
  }
 }
}
```

On iOS, you retrieve the OS language by getting the first item in the **`PreferredLanguages`** collection of the **`NSLocale`** class, which is an object that allows for accessing the local culture information. Now that you have both the implementations, you need a way to consume the information from the shared project regardless of the target operating system. This can be accomplished with the following line:

```
string osLanguage = DependencyService.Get<IDeviceInfo>
().GetSystemLanguage();
```

The **`Get`** method requires the interface as type parameter to retrieve the instance of the appropriate implementation, depending on the current system.

Once you have the instance, you can invoke the members (in this case, **`GetSystemLanguage`**) and get the desired results. In the past, the **`DependencyService`** class was intensively used to access many native features, such as file information, SMS user interface, screen information, and sensors. This is certainly powerful but implies studying and remembering hundreds of native API. Luckily enough, in the last 3 years, Microsoft has been working hard on a library called **`Xamarin.Essentials`**, which simplifies accessing native features from shared code, eliminating the need for **`DependencyService`** in many cases. This is the reason why no other examples are offered about this pattern, in favor of simpler yet effective native access from shared code.

## Cross-platform access to native API: Xamarin.Essentials

**`Xamarin.Essentials`** is a .NET library specifically designed for **`Xamarin.Forms`**; it exposes objects that allow for accessing native, platform-specific features from a shared project without the need for implementing the dependency service pattern. When you create a new **`Xamarin.Forms`** solution, **`Xamarin.Essentials`** is automatically added to the project dependencies. You can quickly verify this by expanding the **`Dependencies`** node in **`Solution Explorer`**. If you need to work on existing projects that do not have this library referenced and want to add it, you can install it via NuGet.

**The purpose of `Xamarin.Essentials` is to make it simpler to access native features that are available on all platforms. For example, you can access the GPS sensor without using the dependency service pattern, but you will be able to leverage sensor features that are commonly available on both Android and iOS (which is usually enough, by the way). If you need further control, you will still need to implement the dependency service.**

**`Xamarin.Essentials`** provides hundreds of objects, so in this section, you will learn how to be ready for the real-world development by focusing on those features that most apps need to implement. You can follow the code examples opening the **`XamarinEssentials`** companion solution in Visual Studio. Remember to add the following directive to the code files that use

**Xamarin.Essentials**: **using Xamarin.Essentials;**.

## Checking the network connection

If your application works over a network, such as the internet or a local intranet, you must implement a check of the connection status before launching any operation that needs a connection. If you do not do this, your application will raise unhandled errors and will provide a very poor user experience. To solve this, you can use the **Connectivity** class and subscribe its **ConnectivityChanged** event, as follows:

```
Connectivity.ConnectivityChanged +=
Connectivity_ConnectivityChanged;
```

You should subscribe to this event in the constructor of a page or in the **App** class definition. The event handler for **ConnectivityChanged** takes an object of type **ConnectivityChangedEventArgs** as the second parameter, which provides information on the network connection status:

```
private async void Connectivity_ConnectivityChanged
  (object sender, ConnectivityChangedEventArgs e)
{
  switch (e.NetworkAccess)
  {
    case NetworkAccess.Internet:
      // App is connected to the Internet:
      break;
    case NetworkAccess.Local:
      // App is connected to a local newtwork
      break;
    case NetworkAccess.ConstrainedInternet:
      // App has limited connection to the Internet
      break;
    case NetworkAccess.None:
      // App is not connected
      break;
    default: // Unknown
      break;
  }
}
```

More specifically, the **NetworkAccess** enumeration returns the status of the network connection. When **None** or **Unknown**, or when your code necessarily requires the Internet and **NetworkAccess** returns a different value, you should inform the user about the lack of connectivity and cancel the requested operation. You can also quickly detect the so-called **connection profile** that allows you to understand if the device is connected via ethernet cable (common on UWP desktop apps), Wi-Fi, cellular network, or Bluetooth. The following code shows an example:

```
 if(Connectivity.NetworkAccess != NetworkAccess.None)
 {
   var profiles = Connectivity.ConnectionProfiles;
   if(profiles.Contains(ConnectionProfile.WiFi) ||
    profiles.Contains(ConnectionProfile.Ethernet))
   {
    // Ethernet or WiFi connection, all good
   }
   else if(profiles.Contains(ConnectionProfile.Cellular))
   {
    // Inform the user about possible charges of their data plan
   }
   else if (profiles.Contains(ConnectionProfile.Bluetooth))
   {
    // Bluetooth is also available
   }
   else
   {
    // Handle unknown status
   }
 }
```

The **`ConnectionProfiles`** collection is populated with a list of values that represent currently active connections on the device. For example, if Wi-Fi, cellular data, and Bluetooth are all enabled on the device, the collection will contain the **`ConnectionProfile.WiFi`**, **`ConnectionProfile.Cellular`**, and **`ConnectionProfile.Bluetooth`** values. The **`ConnectionProfiles`** property is extremely useful; for example, if it only contains the **`Cellular`** value, you should inform your users that they might be charged for the usage of their data plan by their carrier. You are not responsible for their data plan, but you are informing them.

**Tip: On Android, remember to add the `ACCESS_NETWORK_STATE` permission to enable retrieving the connection status.**

## Checking the battery status

One of the biggest differences between mobile app development and desktop or web development is that mobile apps run on devices powered by a battery.

This is an important point because you must handle situations where the battery power is low, especially if your app manages data. By default, both Android and iOS show an alert to the user when the battery level reaches 20%, so it is not necessary to show additional warning messages and bore the user. However, when the battery level is critically down, you must find a way to save data or give the user an option to choose what to do. The **`Xamarin.Essentials`** library provides the **`Battery`** class, which exposes, among others, the **`EnergySaverStatusChanged`** event, which is fired when the device enters the energy saving mode. An example will be provided, but it involves other objects. More specifically, the **`Battery`** class exposes the following members:

- **`ChargeLevel`**, of type double, that returns the current battery level between 0 and 1, where 0 means discharged and 1 means fully charged.
- **`BatteryState`**, an enumeration of type **`BatteryState`**, that returns the state of the battery represented by one of the values described in *Table 13.1*:

| State | Description |
|---|---|
| **`Charging`** | The battery is charging. |
| **`Discharging`** | The battery is discharging. This is also the current status once the device is disconnected from a power source. |
| **`NotCharging`** | The battery is not being charged. |
| **`NotPresent`** | Battery not found (for example, on a desktop computer). |
| **`Unknown`** | The battery status could not be detected. |

***Table 13.1:*** *The BatteryState enumeration*

- **`PowerSource`**, of type **`BatteryPowerSource`**, which returns information about how the device is being powered with one of the values summarized in *Table 13.2*:

| Power source value | Description |
|---|---|
| **`Battery`** | The device is powered by the battery. |
| **`AC`** | The device is connected to an AC unit. |
| **`Usb`** | The device is receiving power through a USB cable. |
| **`Wireless`** | The device is powered via wireless charging. |
| | |

| | |
|---|---|
| **`Unknown`** | The power source could not be detected. |

***Table 13.2:*** *The BatteryState enumeration*

**Tip: On Android, remember to add the `BATTERY_STATS` permission to enable access to the battery info.**

The most efficient way to handle battery status changes is to subscribe to the **`EnergySaverStatus`** event in the constructor of the **`App`** class, as follows:

```
Battery.EnergySaverStatusChanged +=
Battery_EnergySaverStatusChanged;
```

The event handler needs to check whether the battery level is less than 20% and if energy saving is on:

```
private void Battery_EnergySaverStatusChanged(object sender,
   EnergySaverStatusChangedEventArgs e)
{
  MessagingCenter.Send(this, “BatteryEvent”, e.EnergySaverStatus
  ==
       EnergySaverStatus.On && Battery.ChargeLevel <= 0.2);
}
```

As the **`App`** class cannot do anything on the user interface, when the event is intercepted and handled, a broadcast message is sent to any object that has subscribed for it. For example, a view could subscribe for such a message, as follows:

```
MessagingCenter.Subscribe<App, bool>(this, “BatteryEvent”,
   ManageBatteryLevelChanged);
```

Additionally, in the **`ManageBatteryLevelChanged`** method, the view could save data locally or do anything else that could be required in order to prevent data loss in case the battery suddenly goes down very fast. Remember that handling the battery status with applications that allow users to enter data is not an option. You are responsible for keeping user data in a good state and make them safe at all times.

## Sending emails and SMS messages

It is not uncommon for mobile apps to offer the possibility to send emails and SMS messages. An example is providing contact options. This section describes both scenarios and shows how easy it is to send both kind of messages with **`Xamarin.Essentials`**.

## Sending emails

**`Xamarin.Essentials`** exposes the **`Email`** class, which exposes the **`ComposeAsync`** method, among others. When invoked, this method launches the system default email client, so there is no need to build your own user interface. This is the recommended approach for at least two reasons:

- Users are comfortable with an email client they already know.
- You do not take responsibility over security of the email exchange.

In order to send an email, you first create an instance of the **`EmailMessage`** class and populate its self-explanatory properties, as follows:

```
EmailMessage message = new EmailMessage();
message.Subject = “Contact request”;
message.To = new List<string> { “support@yourcompany.com” };
message.Cc = new List<string> { “marketing@mycompany.com” };
message.BodyFormat = EmailBodyFormat.PlainText;
message.Body = “We would need to meet your developers to suggest
features.”;
```

Note how you can add multiple recipients via a **`List<string>`**. Note how the **`EmailBodyFormat`** enumeration allows for specifying **`PlainText`** or **`Html`** as the email format. In the case of HTML, you will need to pass the body as a string containing HTML. The last step is the following simple invocation:

```
await Email.ComposeAsync(message);
```

This will launch the default email client on the target device, supplying a pre-configured email message. It is also possible to specify attachments. Every attachment is represented by an instance of the **`EmailAttachment`** class, which is added to the **`Attachments`** collection as follows:

```
string attachmentPath = Path.Combine(Environment.GetFolderPath(
        Environment.SpecialFolder.MyPictures),
        “attachedImage.jpg”);
EmailAttachment attachment = new
EmailAttachment(attachmentPath);
message.Attachments = new List<EmailAttachment>();
message.Attachments.Add(attachment);
```

In the previous code, an image file residing in the local photo gallery is attached to the email message. It will be responsibility of the client to upload and send the attachment along with the message.

## Sending SMS messages

In the era of free messaging apps working over the internet, SMS messages might seem anachronistic. However, there are plenty of reasons to still use them, and the most important one is that they work even if there is no internet connection. The **`Sms`** class allows for sending SMS messages, and you have a few options. The following line of code shows the system user interface to send SMS messages with empty fields:

```
Sms.ComposeAsync();
You could also prepare an SMS message as follows:
var message = new SmsMessage();
message.Body = “Text of the message”;
message.Recipients = new List<string>{“Alessandro Del Sole”};
Sms.ComposeAsync(message);
```

In this second example, you create an instance of the **`SmsMessage`** class, assigning the text for the message to the **`Body`** property and the list of recipients to the **`Recipients`** property. The latter is any collection that implements **`IEnumerable<string>`**, and obviously, you must ensure that the name of the recipient matches a name in the contact list of the device. You could pass the body and the list of recipients to the constructor of the **`SmsMessage`** class, but then you would be forced to pass both. With the proposed approach, you can just create the body and make the user select recipients from the more convenient system user interface.

## Opening contents

Another common option with mobile apps is opening external contents, such as websites or applications on the device. This section describes how to handle both scenarios.

## Opening the web browser

For websites or any web hosted content, you can open the default web browser. Though **`Xamarin.Forms`** provides the **`WebView`** to display web and HTML contents, this is not the recommended approach at times. For example, if your app needs to display advice from a medical website, opening the content with a **`WebView`** gives the user the perception that the information is provided by you. Instead, the information is provided by third parties, and this should be very clear to the user, especially when you display sensitive information such as medical data and payout information on e-commerce sites. Opening the default web browser is possible with just one line of code, as follows:

```
await Browser.OpenAsync("https://www.microsoft.com");
```

The **`OpenAsync`** method from the **`Browser`** class simply takes the target address as the parameter. For example, if no changes have been made, this will open Chrome on Android and Safari on iOS.

## Opening default apps

It is also possible to open specified resources with the dedicated, default apps on the device. For instance, you can open a remote PDF document with the default viewer on the target device as follows:

```
string url = "https://www.someproducts.com/usermanual.pdf";
if(await Launcher.CanOpenAsync(url))
  await Launcher.OpenAsync(url);
```

The **`Launcher`** class exposes the **`CanOpenAsync`** method, which you should first invoke to make sure that the device has viewers for the specified resources. If it returns true, you can then invoke **`OpenAsync`** to open the resource.

## Storing user preferences

You already got an example of storing local preferences and settings in *Chapter 11: Managing the Application Lifecycle*, so you will get more details in this section. The **`Preferences`** class allows for locally storing and retrieving key/value pairs, where the key uniquely identifies the information, and the value can be one of the .NET primitive types. The following code shows an example:

```
// Stores an integer
Preferences.Set("NumberOfTapsPerAction", 2);
// Stores a DateTime
Preferences.Set("LastDateTimeAccess", DateTime.Now);
// Checks if the key exist
if (Preferences.ContainsKey("NumberOfTapsPerAction"))
  // Returns the value for the key, and a default value
  // if the key is not found
  Preferences.Get("NumberOfTapsPerAction", 0);
```

The **`Get`** method allows you to specify a default value in case the key does not exist, so checking with **`ContainsKey`** is not mandatory. However, this method can just be useful to detect the existence of the key. It is also possible to remove individual keys or everything with the **`Remove`** and **`Clear`** methods, respectively, as follows:

```
// Remove an individual key
Preferences.Remove(“NumberOfTapsPerAction”);
// Remove all preferences
Preferences.Clear();
```

On Android, preferences are stored in the system’s shared preferences (**https://developer.android.com/training/data-storage/shared-preferences**). On iOS, preferences are stored in the User Defaults system database (**https://docs.microsoft.com/en-us/xamarin/ios/app-fundamentals/user-defaults**). Preferences are a convenient way to quickly store small pieces of information, but they are not secured. For a more secure way to store local information, you can use the secure storage, which will be discussed in the next section.

**When an app is uninstalled, local preferences are also removed.**

## Storing secure settings

Sometimes, it is necessary to store settings locally, in a secured way; for example, in the case of passwords or any sensitive information that could identify the user. For this purpose, both Android and iOS provide secure local storages. On Android, this is represented by the KeyStore (**https://developer.android.com/training/articles/keystore**), whereas on iOS, it is represented by the KeyChain (**https://developer.apple.com/documentation/security/keychain_services**). There are important differences between preferences and secure storage:

- Preferences are stored in a space that is reserved for the app and are removed when the app is uninstalled.
- Secure storages are at the system level, which means information stored in the secure storage is not removed when an application is uninstalled and require a few more checks from the developer.
- Preferences support any .NET primitive type, but the secure storage only supports the **`string`** type.

Having that said, **`Xamarin.Essentials`** offers the **`SecureStorage`** class, which you use as follows:

```
// Add a key/value pair to the secure storage
await SecureStorage.SetAsync(“UserID”, “123456”);
// Get a key/value pair from the secure storage
string userIDValue = await SecureStorage.GetAsync(“UserID”);
// Removes the specified key from the secure storage
SecureStorage.Remove(“UserID”);
```

In summary, **`SetAsync`** saves the specified string value under the assigned key, whereas **`GetAsync`** retrieves the string value of the specified key and returns null if no key is found. The **`Remove`** method removes the specified key from the storage. Another method, **`RemoveAll`**, removes all the keys from the secure storage; you should completely avoid using it.

**Tip: You can certainly trust the secure storage. However, if you want an additional level of security, you can use one of the .NET encryption algorithms to encrypt a string and save this one to the secure storage, and then you can decrypt the string when you read it back from the secure storage. This can be a good approach when your app manages very sensitive data, such as bank accounts or medical data.**

## Secure settings with app version tracking

As mentioned earlier, keys are not removed from the secure storage when an app is uninstalled because the secure storage is shared at the OS level. An app can be uninstalled and then reinstalled at a later stage; so, if it uses the secure storage, it is important to provide a fresh environment. To accomplish this, you can combine the **`SecureStorage`** class with the **`VersionTracking`** class (this is also offered by **`Xamarin.Essentials`**). The latter provides members that allow for version tracking, including a way to understand whether the app is running for the first time on the device. Consider the following code:

```
private async Task<bool> IsAppRunningForFirstTime()
{
 if (VersionTracking.IsFirstLaunchEver)
 {
  string userID = await SecureStorage.GetAsync(“userID”);
  if (userID != null)
   SecureStorage.Remove(“userID”);
  return true;
 }
 return false;
}
```

The **`VersionTracking.IsFirstLaunchEver`** method returns true if the app is running on the device for the very first time. In this case, the goal would be to provide a fresh environment so that the code checks whether the specified key already exists and removes the key if so. In order to enable version tracking, you need to add the following line of code to the constructor of the `App` class:

```
VersionTracking.Track();
```

Other useful members from the `VersionTracking` class are listed here:

- `IsFirstLaunchForCurrentVersion` returns true if the current version of the app is running for the first time on the device.
- `IsFirstLaunchForCurrentBuild` returns true if the current build of the app is running for the first time on the device.
- `VersionHistory` is an `IEnumerable<string>` object that contains the list of versions of the app that have been installed on the device.
- `CurrentVersion` contains the version number of the app.
- `CurrentBuild` contains the build number of the version.

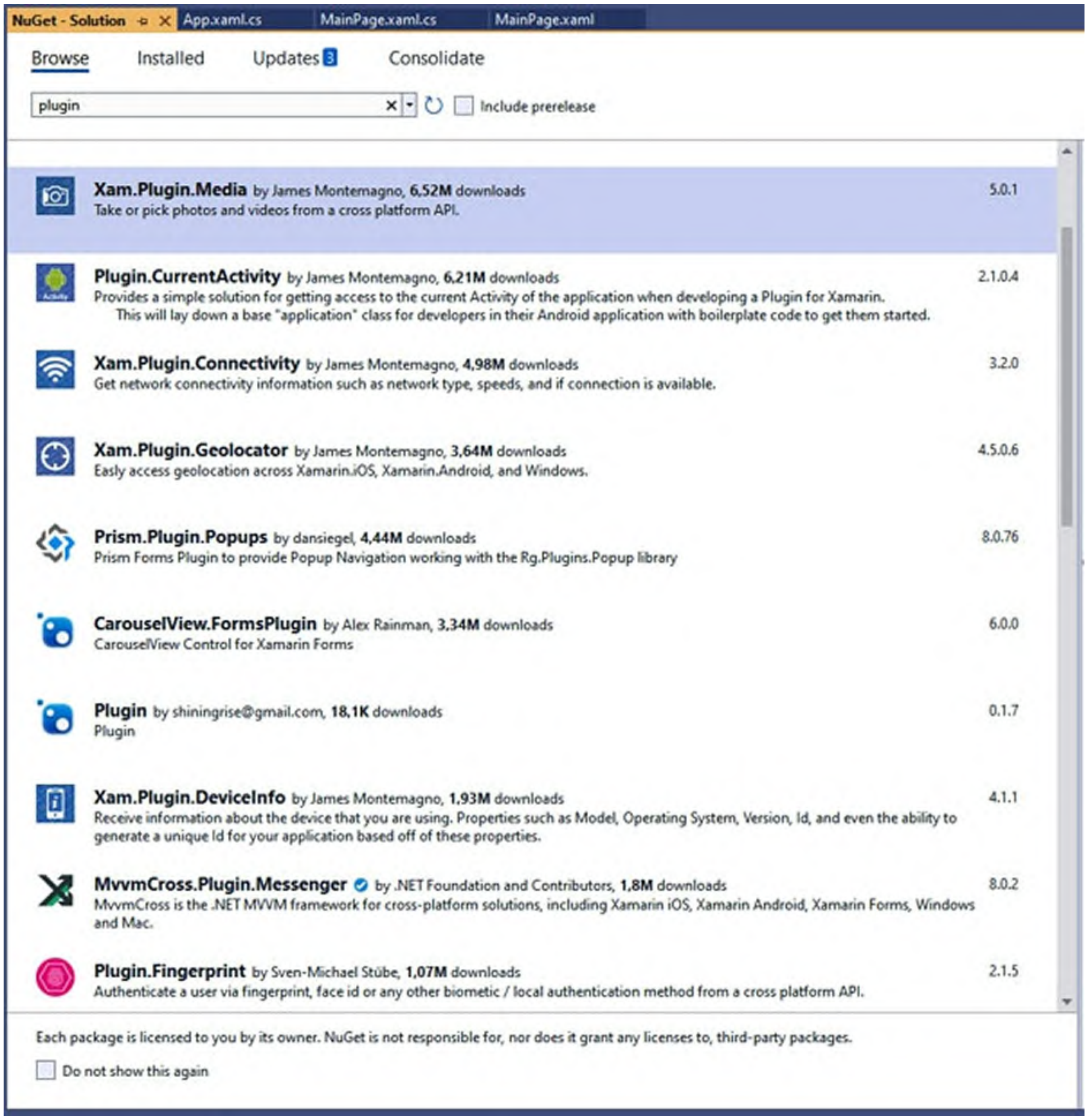
The `VersionTracking` class can be extremely useful when you want to implement a mechanism for providing mandatory updates or offer updated content that needs to be downloaded before people can use the app.

## More essentials API

`Xamarin.Essentials` has dramatically improved, especially over the last year: many new cross-platform APIs have been added to the library. This chapter has described objects that are of common usage in a variety of applications, but there is certainly much more. For example, `Xamarin.Essentials` makes it simpler to access hardware devices such as gyroscope, accelerometer, orientation sensor, and GPS services. The full list of supported APIs, which grows with every new release, with code examples is available in the official Microsoft documentation (**https://docs.microsoft.com/en-us/xamarin/essentials/**). In addition, keep in mind the following three considerations every time you implement features that collect personal data, such as geolocation: you must explicitly ask the user for permission to enable sensors and collect data, you must supply a clear privacy policy that explains why you need one or more sensors enabled and which data you collect (and what you do with it), and you must provide users with a way to remove their consent at any time. These are mandatory things to do according to the publishing guidelines for both the Apple Store and the Google Play. Moreover, most countries have laws that protect users' privacy, so whenever you are in doubt, check with a lawyer.

## Hints about plugins

The goal of unifying access to native APIs into shared code has always been important to the developer community worldwide. So, during the years, individual developers, including Microsoft employees, have built the so-called **plugins**, libraries that enable for accessing native iOS and Android features from the `Xamarin.Forms` project. `Xamarin.Essentials` was actually born long after plugins, with the same idea in mind, but with many features in one place. A lot of plugins still exist, and many of them can still be found in real-world projects. They can be installed from NuGet, and almost all of them are free. *Figure 13.3* shows a list of plugins in NuGet:



**Xam.Plugin.Media** by James Montemagno, 6,52M downloads — 5.0.1
Take or pick photos and videos from a cross platform API.

**Plugin.CurrentActivity** by James Montemagno, 6,21M downloads — 2.1.0.4
Provides a simple solution for getting access to the current Activity of the application when developing a Plugin for Xamarin. This will lay down a base "application" class for developers in their Android application with boilerplate code to get them started.

**Xam.Plugin.Connectivity** by James Montemagno, 4,98M downloads — 3.2.0
Get network connectivity information such as network type, speeds, and if connection is available.

**Xam.Plugin.Geolocator** by James Montemagno, 3,64M downloads — 4.5.0.6
Easly access geolocation across Xamarin.iOS, Xamarin.Android, and Windows.

**Prism.Plugin.Popups** by dansiegel, 4,44M downloads — 8.0.76
Prism Forms Plugin to provide Popup Navigation working with the Rg.Plugins.Popup library

**CarouselView.FormsPlugin** by Alex Rainman, 3,34M downloads — 6.0.0
CarouselView Control for Xamarin Forms

**Plugin** by shiningrise@gmail.com, 18,1K downloads — 0.1.7
Plugin

**Xam.Plugin.DeviceInfo** by James Montemagno, 1,93M downloads — 4.1.1
Receive information about the device that you are using. Properties such as Model, Operating System, Version, Id, and even the ability to generate a unique Id for your application based off of these properties.

**MvvmCross.Plugin.Messenger** by .NET Foundation and Contributors, 1,8M downloads — 8.0.2
MvvmCross is the .NET MVVM framework for cross-platform solutions, including Xamarin iOS, Xamarin Android, Xamarin Forms, Windows and Mac.

**Plugin.Fingerprint** by Sven-Michael Stübe, 1,07M downloads — 2.1.5
Authenticate a user via fingerprint, face id or any other biometric / local authentication method from a cross platform API.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

***Figure 13.3:*** *A list of Xamarin.Forms plugins*

For example, the **`Xam.Plugin.Media`** plugin makes it easy to capture pictures and videos from the device's camera in a cross-platform approach, whereas the **`Plugin.Fingerprint`** plugin enables quick implementation of biometric authentication. Every plugin has its own documentation, which you can find by clicking on the link that appears every time you select a plugin in the list. Some plugins are no longer maintained, and some of them have been embedded into **`Xamarin.Essentials`**, so you should avoid plugins when possible and rely on **`Xamarin.Essentials`**. If you need to implement functionalities that **`Xamarin.Essentials`** does not support but plugins do, make sure they are maintained by checking the release history on the plugin support page.

## Conclusion

The biggest benefit of **`Xamarin.Forms`** is that it provides a unified, cross-platform way to access features that are available on different systems. However, sometimes you need to access native views and device features. When it comes to views, you can use custom renderers for deep view customizations and behavioral changes, whereas you can use effects for customizations that do not require changing the behavior of the view. You can also embed native views directly in your XAML markup, or you can use platform-specifics to assign native properties to **`Xamarin.Forms`** views. When it comes to accessing device and system features, you can use the **`DependencyService`** class and the same-named pattern. However, a more modern approach is to use the **`Xamarin.Essentials`** library, which provides a large set of .NET API to access native system features from **`Xamarin.Forms`** code. The discussion about accessing native APIs also marks the end of the technical chapters of this book. Now that you have all the necessary knowledge, you need to either start working on new projects or on existing projects. In the next two chapters, you will find a lot of useful advice about finding a good job and how to stay successful in the software industry.

## Key terms

- **Thread-safe**: Code that lives inside a thread, along with the resources it uses.
- **Renderer**: A C# class that makes it possible to expose a native Android and iOS view to Xamarin.Forms.
- **Platform-specifics**: A feature that allows assigning native properties to **`Xamarin.Forms`** views directly in XAML.
- **Dependency service**: A **`Xamarin.Forms`** programming pattern based on native implementations of an interface, whose members are invoked in the shared project regardless of the target system.

## Suggested readings

- Custom renderers documentation (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/**).
- Effects documentation (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/effects/**).
- Dependency service documentation (**https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/dependency-service/introduction**).
- Xamarin.Essentials documentation (**https://docs.microsoft.com/en-us/xamarin/essentials/**).

# CHAPTER 14
# Finding a Job

## Introduction

As a Xamarin jobseeker, you will probably want to practice what you have learned from a technical point of view by getting a job that requires you to use your skills. In this chapter, you will find many useful suggestions about searching for jobs that are related to your skills and profile, and you will also find information about preparing for job interviews. Obviously, these suggestions not only apply to Xamarin-related jobs but to various positions in information technology. Finally, as you might have noticed, the word suggestions is used because there are literally no rules. Every person can have a different approach to searching for a job and applying for a position, but some general recommendations can be useful to everyone.

## Structure

In this chapter, we will cover the following topics:

- Preparing your resume
- Finding jobs, the modern way: Using LinkedIn
- A step forward: Attracting jobs
- Preparing for job interviews

## Objectives

After completing this chapter, you will have learned about modern ways for finding jobs as a mobile app developer working with Xamarin. You will have discovered ways to not only search for jobs but also to attract them.

## Preparing your resume

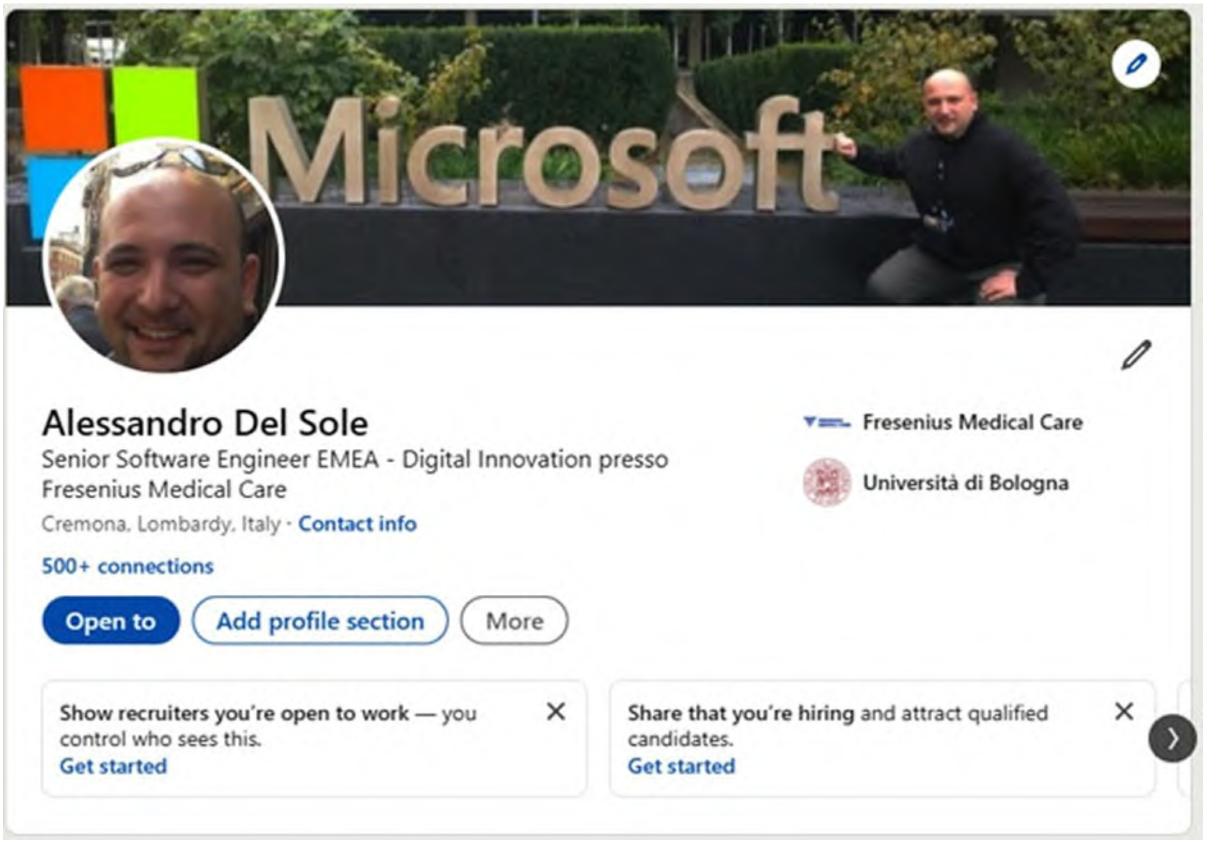
The resume is a crucial document when it comes to finding a job. It summarizes who you are, what your experience is, what studies you have completed, and what value you can bring to the company that is hiring you. On the internet, you can find thousands of resume templates and choose one that best fits your taste; however, it is important that you make the following considerations:

- **Keep it concise**: Two pages are more than enough. A recruiter needs to focus on key information, not on the story of your life (at least in a first contact).
- **Add a profile picture**: It gives the perception of reading the resume of a real person, especially in today's times of remote connections.
- **Contact information**: Add your basic contact information and date of birth.
- **Add an honest and realistic list of your technical skills**: This is the first thing that an employer will look at. Do not lie when it comes to this.
- **Add your highest school degree only**: If you have attended university, add your university degree; otherwise, add your high school degree. There is no need to add all your scholar story, unless it is relevant to the position you apply for.
- **Certified expertise**: Add any certifications or courses that are relevant to the position you are applying for.
- **Existing experience**: If you have prior work experience, add a summary with a short description of what you have done in each job.
- **Make it consistent**: If you apply for a job that requires experience with Xamarin, only add details of your skills about Xamarin, .NET and technologies you generally use with Xamarin (for example, Web API, SQL, Azure). For other skills, write a short summary in one line.
- **Online presence**: Add links to professional social media profiles, such as LinkedIn, and to public code repositories you might have, such as GitHub. For some companies, this really makes a difference.
- **Personal life**: Write one or two lines about your personal interests and hobbies outside of work.

Most employers and recruiters know you will send your resume to multiple companies, so it is generally appreciated when they understand you wrote one that is tailored to their needs. In general, always be honest. Do not say you have experience in something only to address a company's requirements and attract interest. Saying something false is not the best way to start establishing a trustworthy relationship between you and a company. In addition, do not over evaluate skills that are the basis for working in the IT world. For example, even if you are the best in using Excel, do not write that you are Excel's god. Simply mention that you have strong knowledge of the Office package. Honesty and simplicity are keys to a good start. Finally, update the date on your resume every time you send it to a different company, print it, add your signature, scan it, and send it as a PDF. Updating the date and adding a signature means you want to demonstrate that they can trust you. Producing a PDF document is important because it can be read even on mobile devices, so any employer is free to read it anytime and anywhere.

## Finding jobs, the modern way: Using LinkedIn

The classic approach to finding a job is looking for postings by companies on newspapers, university bulletin boards, or sending your resume directly to companies you know. This is still something you can do, but the appropriate approach to finding a job in the IT industry today is using LinkedIn (**https://www.linkedin.com**). For years, LinkedIn has been the most important and the most popular professional social network. So, if you do not already have an account on LinkedIn, you should create one (**https://www.linkedin.com/signup**). When you create an account, you will be able to enter all your contact and personal details, and it is recommended that you add a profile picture. There are a few sections in your profile that you should always keep up to date; you can access these by clicking on **`Me`** in the upper-right corner of the page and then on **`View Profile`**. The first section contains your picture and a summary of your current activities, as shown in *Figure 14.1*, where you can see the profile page for the author of this book.



Alessandro Del Sole

Senior Software Engineer EMEA - Digital Innovation presso Fresenius Medical Care

Cremona, Lombardy, Italy · Contact info

500+ connections

Fresenius Medical Care

Università di Bologna

Show recruiters you're open to work — you control who sees this.

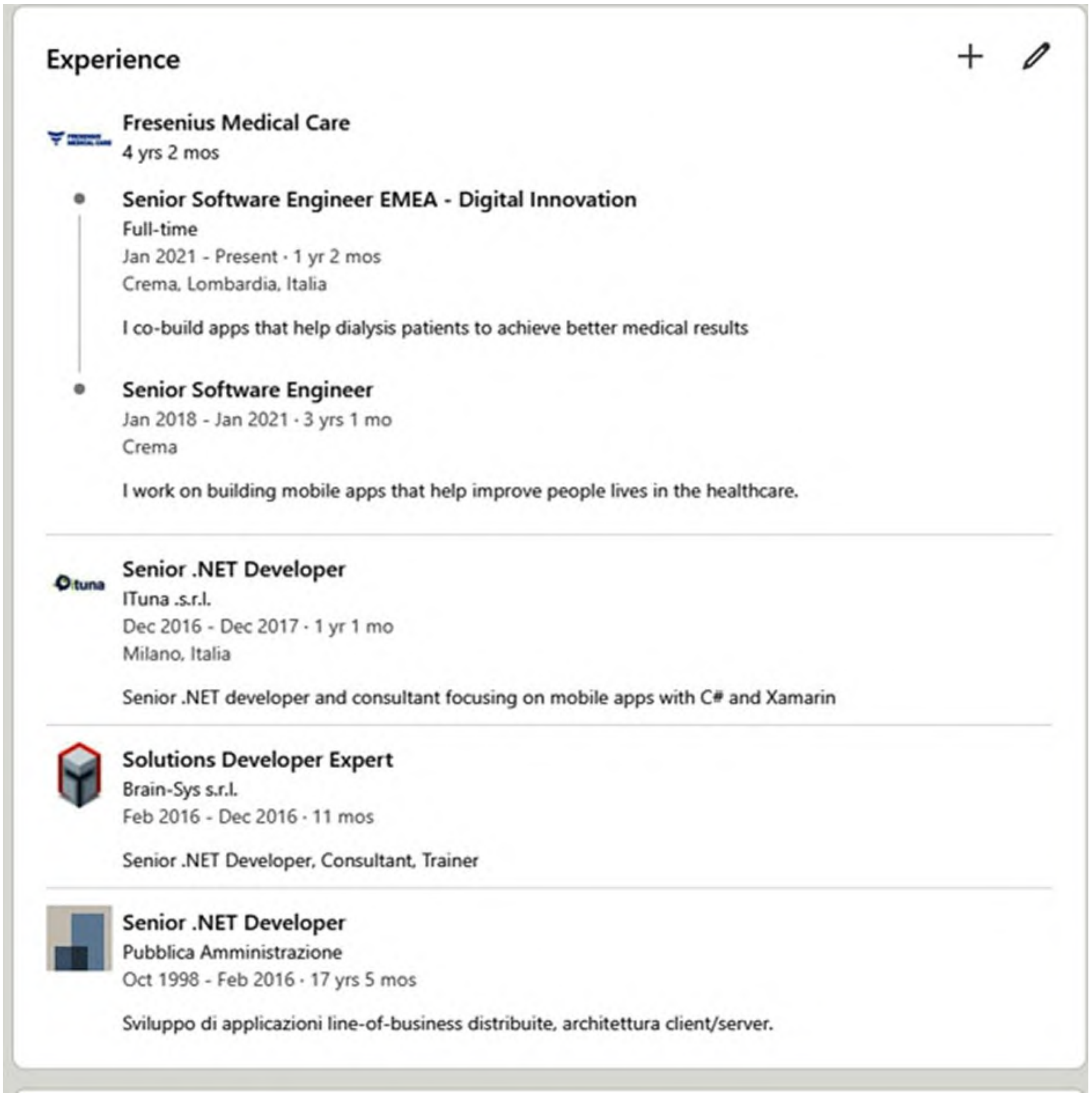
Get started

Share that you're hiring and attract qualified candidates.

Get started

***Figure 14.1:*** *Summary of your LinkedIn profile*

The next part you should keep up to date is your work experience, if any, even if you have not been working for a while. *Figure 14.2* shows an example:


Experience

Fresenius Medical Care
4 yrs 2 mos

Senior Software Engineer EMEA - Digital Innovation
Full-time
Jan 2021 - Present · 1 yr 2 mos
Crema, Lombardia, Italia

I co-build apps that help dialysis patients to achieve better medical results

Senior Software Engineer
Jan 2018 - Jan 2021 · 3 yrs 1 mo
Crema

I work on building mobile apps that help improve people lives in the healthcare.

Senior .NET Developer
ITuna .s.r.l.
Dec 2016 - Dec 2017 · 1 yr 1 mo
Milano, Italia

Senior .NET developer and consultant focusing on mobile apps with C# and Xamarin

Solutions Developer Expert
Brain-Sys s.r.l.
Feb 2016 - Dec 2016 · 11 mos

Senior .NET Developer, Consultant, Trainer

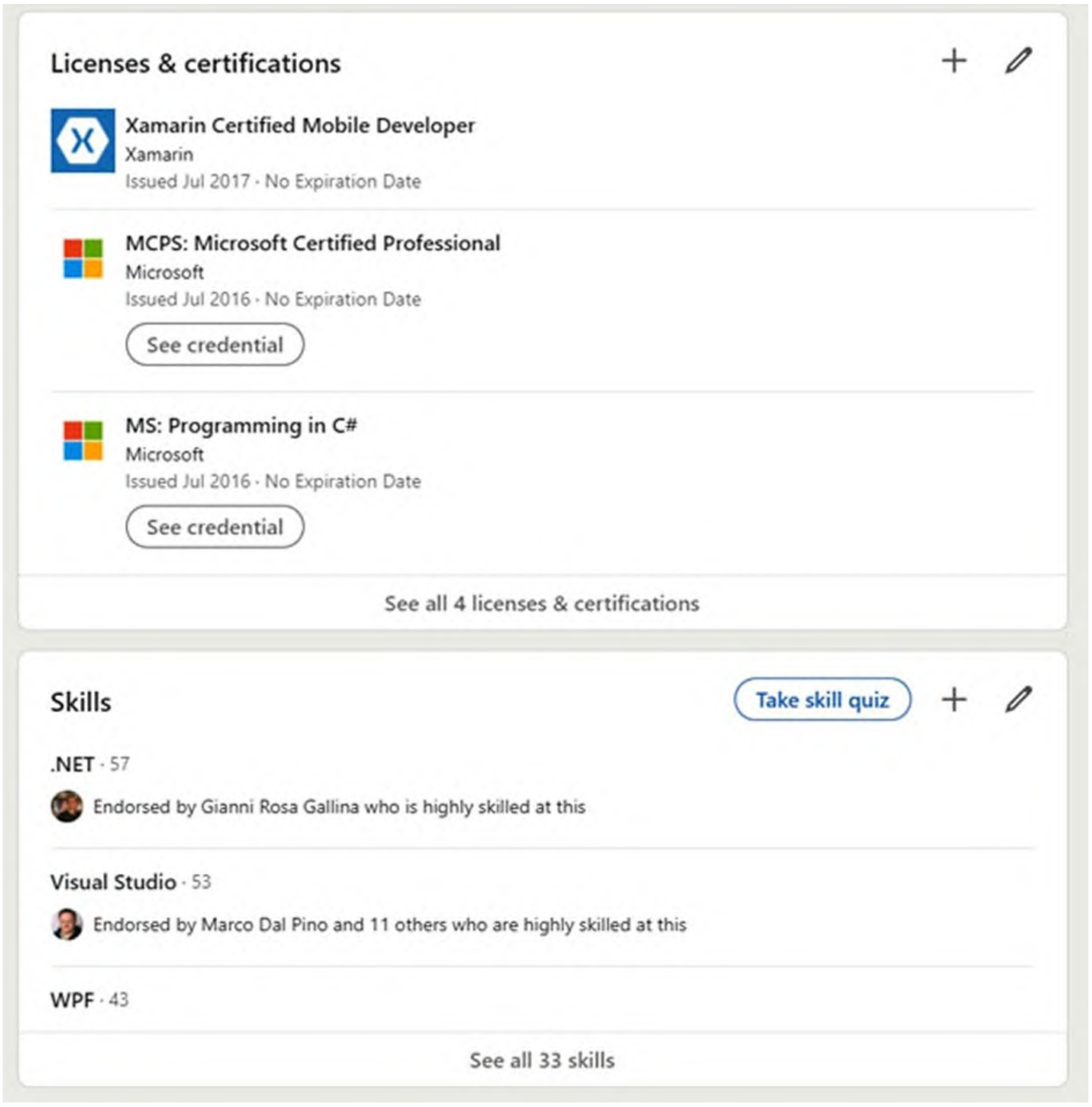
Senior .NET Developer
Pubblica Amministrazione
Oct 1998 - Feb 2016 · 17 yrs 5 mos

Sviluppo di applicazioni line-of-business distribuite, architettura client/server.

***Figure 14.2:*** *Keeping your work experience up to date*

**Tip: Every section of the LinkedIn profile can be updated by clicking on the pencil icon.**

Next, there are other two key sections: **`Licenses & certifications`** and **`Skills`**. Here, you can add any official certifications you have gained with studies and exams, and you can add a list of your technical skills (see *Figure 14.3*):


Licenses & certifications

Xamarin Certified Mobile Developer
Xamarin
Issued Jul 2017 · No Expiration Date

MCPS: Microsoft Certified Professional
Microsoft
Issued Jul 2016 · No Expiration Date
See credential

MS: Programming in C#
Microsoft
Issued Jul 2016 · No Expiration Date
See credential

See all 4 licenses & certifications

Skills
Take skill quiz

.NET · 57
Endorsed by Gianni Rosa Gallina who is highly skilled at this

Visual Studio · 53
Endorsed by Marco Dal Pino and 11 others who are highly skilled at this

WPF · 43

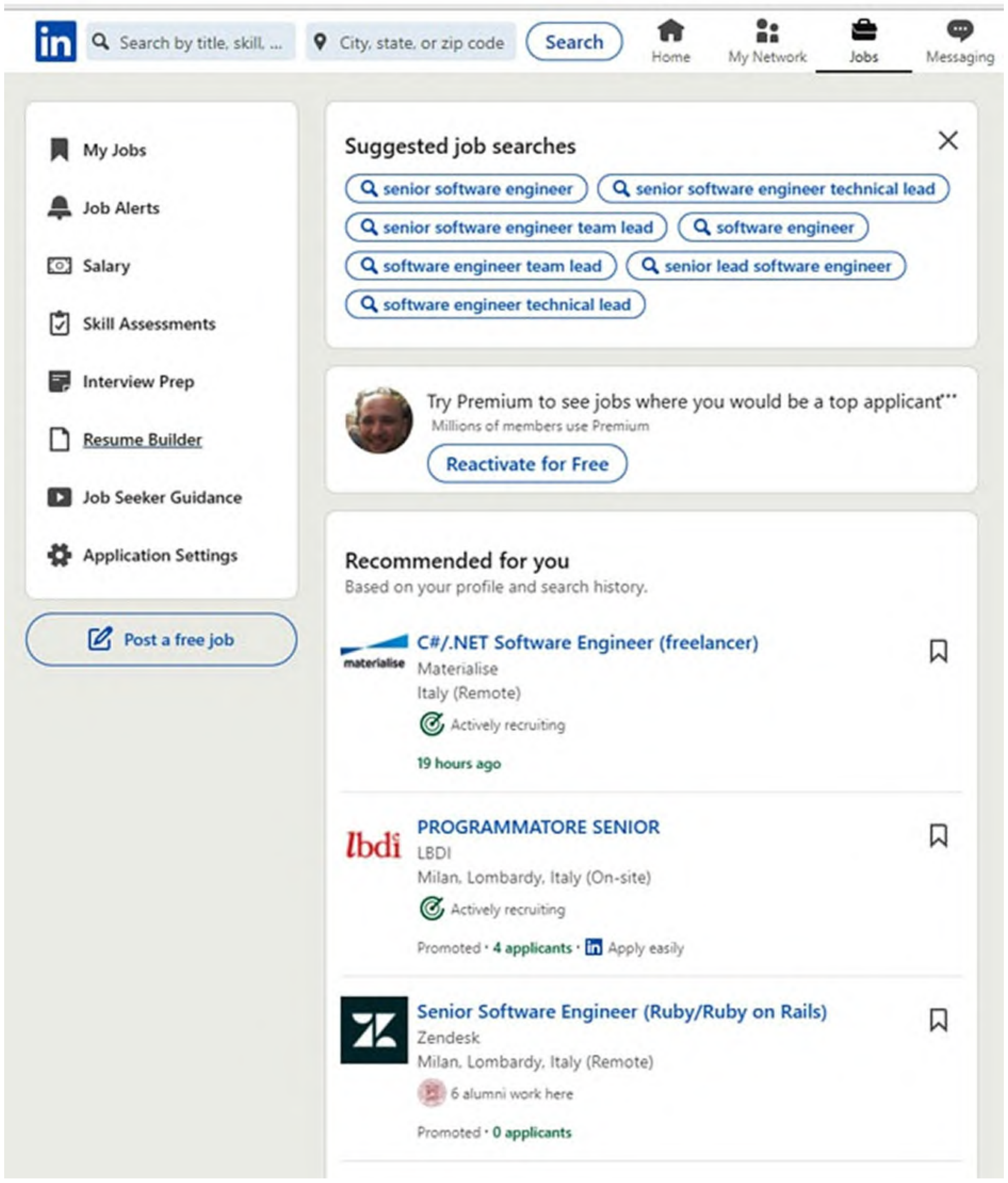
See all 33 skills

***Figure 14.3:*** *Certifications and skills*

For the skills, you can add what you feel more prepared about, and other people on LinkedIn who know you can confirm what you say. This is beneficial because any recruiter looking at your profile can see how strong your knowledge is on specific topics, confirmed by other professionals. This is an important point, and it needs specific considerations, which will be covered in the next section, *A step forward: Attracting jobs*.

## Searching for a job

If you click on **`Jobs`** on the toolbar at the top of the page, you will be able to search for a job on LinkedIn. *Figure 14.4* shows an example:



Suggested job searches

senior software engineer | senior software engineer technical lead | senior software engineer team lead | software engineer | software engineer team lead | senior lead software engineer | software engineer technical lead

Try Premium to see jobs where you would be a top applicant**

Millions of members use Premium

Reactivate for Free

Recommended for you

Based on your profile and search history.

C#/.NET Software Engineer (freelancer)
Materialise
Italy (Remote)
Actively recruiting
19 hours ago

PROGRAMMATORE SENIOR
LBDI
Milan, Lombardy, Italy (On-site)
Actively recruiting
Promoted · 4 applicants · Apply easily

Senior Software Engineer (Ruby/Ruby on Rails)
Zendesk
Milan, Lombardy, Italy (Remote)
6 alumni work here
Promoted · 0 applicants

***Figure 14.4:*** *Searching for a job*

LinkedIn will propose a series of job postings according to the skills, certifications, location, and current position you have added to your profile.

In addition, you can manually search for open jobs by typing in the search bar in the upper-right corner or by selecting a location (entering remote is also supported). On the left side of the page, there is a menu with options that will help you get the most out of LinkedIn. For example, you can upload your resume to your profile, or you can watch an instructional video about the best practices for finding jobs. You can click on an open position to see its full details. If you think it is a good opportunity for you, click on the **`Apply`** button and follow the steps. As you can imagine, with LinkedIn, you have an infinite number of possibilities, and you will be able to focus on open positions that better fit with your skills and expectations. However, LinkedIn has another benefit: being a social media platform, it also allows recruiters to find the best people for their needs. This is the topic of the next section.

## A step forward: Attracting jobs

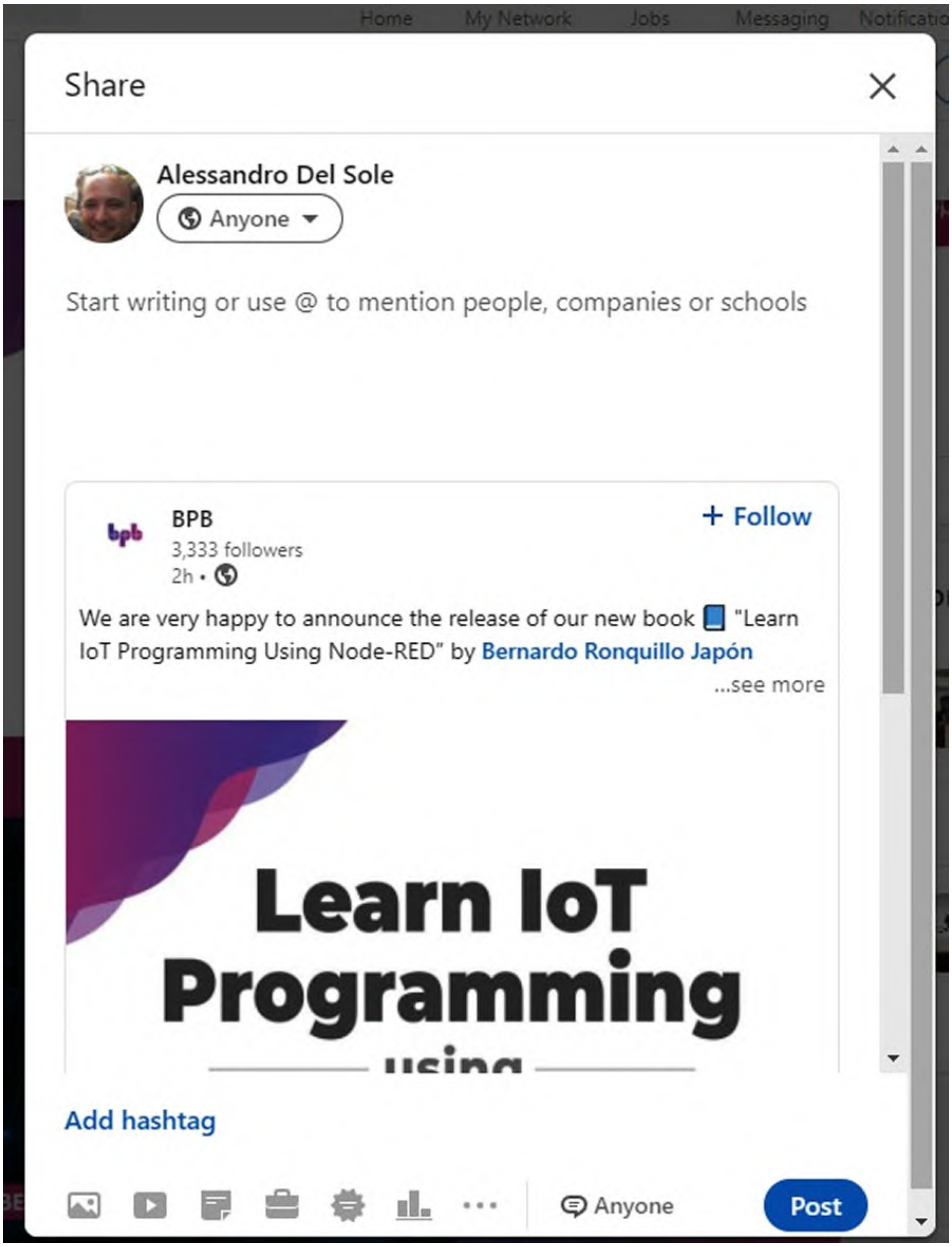
One of the biggest benefits of using LinkedIn is that it makes it easy to attract jobs. Attracting a job means that recruiters will discover your profile based on what you publish and share, and they will be able to connect with you to offer a job if your profile matches their needs. This means you might be able to find a job without even applying for a position. However, this can only happen if the LinkedIn search algorithm can find your profile when recruiters search, which does not happen with no effort. The next sections discuss how you can attract jobs, opening your mind to a wider range of possibilities. However, always keep in mind: success does not come for free; it requires commitment.

## Connecting with the right contacts

LinkedIn can be of great help in finding a job, but this requires building an appropriate contact network. Remember that this is not a fun social media, so you should not focus on adding to your network any person you have met in your life. To start, send contact requests only to people who you know from a professional point of view. If you are new to the professional world, or as a second step, if you already added people to your network, in the search bar of LinkedIn enter terms related to the technology you are interested in, for example, Xamarin. A list of people, companies, and groups matching your criterion will appear. Start following companies and groups, and then look at the people who might have an interest to connect with you, such as common interests, companies you both follow, and so on. If you have a new account, it might happen that person you want to connect to will not accept your invite, but this should not be discouraging. This is something that the next sections will help improve. As your network and activities on LinkedIn grow, your rule should be connecting with people with the same technical interests as yours and with people from the human resources departments. You will discover on your own that this is not difficult at all.

## Sharing contents from others

The key to attracting a job is making your profile more discoverable. This can be easily accomplished by writing posts about your interests or sharing content. As a first step, you can share contents published by other people, groups, or companies that you have started following with the suggestions explained in the previous paragraphs. Locating content that can be shared is easy: you can find it when scrolling your timeline or by opening the profile page of a company or person you follow. For each piece of content they have posted, LinkedIn enables a **`Share`** button. *Figure 14.5* shows how to share a post from a company:


Share

Alessandro Del Sole

Anyone

Start writing or use @ to mention people, companies or schools

BPB
3,333 followers
2h •

We are very happy to announce the release of our new book 📘 "Learn IoT Programming Using Node-RED" by Bernardo Ronquillo Japón
...see more

***Figure 14.5:** Sharing content from companies or other people*

This simple action has several outcomes:

- The author of the content can see that you have shared it, so they might follow you back.

- The content you have shared appears to other people in your network, so they can see what you are more interested in.
- The LinkedIn search algorithm starts understanding your interests better and suggests your post to other people with the same interests, making your profile more easily discoverable.

However, this is not enough to attract a job. Recruiters and other people are interested in what you have to say, which means sharing your knowledge.

## Sharing your knowledge

You are one in a million. Making yourself and your profile easily discoverable is the key to attracting a job. The best way to do this is to share your knowledge, which is fundamental in the IT world today. Publicly showing what you know is the best way to make yourself discoverable and the best way to create trust in your knowledge (if what you write is technically accurate). Sharing your knowledge means writing articles, recording instructional videos, starting a technical blog, and/or publishing open-source projects. You do not need to become a book author or a conference speaker; there are things you can do from the comfort of your home that will make a difference. Sharing content online already makes you discoverable, and LinkedIn can help boost your visibility by publishing posts that make your network know what you are doing for the developer community.

### Writing articles and other content

The first type of content you can publish is technical articles. As a new publisher, the suggestion is to start with websites that offer free space and have reviewers. The best place to start is C# Corner (**https://www.c-sharpcorner.com/**). This is one of the most popular websites that allow the publishing of technical articles, and every time you publish something, your content is first reviewed for approval by a person in their team. You do not need to write long pieces; the important thing is that you write accurate content. The more you publish accurate content, the more your reputation grows. What you can do is study a topic or take examples from your real-life experience and write an article on it. When you article is live, go to LinkedIn, create a new post, share the link to your article and add a caption that says that you have published a new article. LinkedIn also allows you to add hashtags to posts, which will help make them more visible. The same concepts apply to recording instructional videos that you can publish on YouTube. This probably requires more self-confidence, but it will be of help in the long term.

## Joining online communities

Online communities are another great place to share your knowledge, and they offer another possibility to find jobs outside of LinkedIn. These communities often rely on a website with content, forums, and other options to participate and share your knowledge. There are thousands of communities in the World, so try to run a search on Google to see if any is located close to your area of living. In fact, most times, they also organize in-person events, which is important to strengthen your relationships with others.

## Sharing your code

Sharing the code you write should not be underestimated. For example, you can create small libraries or applications that demonstrate how to accomplish a few tasks and publish the source code to popular online websites, such as GitHub (**https://github.com**) or Bitbucket (**https://bitbucket.org**). This helps improve your reputation and discoverability, and there are some companies that require public code repositories as part of your resume. The steps to publish code on an online repository are out of the scope of this book, but there are plenty of online guides that explain how to accomplish this. The important thing is that this is another kind of content that you can share with LinkedIn posts, where you can add the link to the resource and a brief description (do not forget to use hashtags).

**Tip: Websites such as C# Corner and GitHub are so popular that not only is their content easy to find through a Google search, but they are also considered by millions of developers as container of resources with their own life, and that can boost your visibility even without LinkedIn. Especially if you are new to software development, it would be a good idea to find someone who can review your articles and code before they go live, until you are confident with your skills.**

### Final considerations about LinkedIn

If you follow the suggestions provided so far, your LinkedIn profile will stay tremendously active and will be easy to discover among other millions of profiles. This will make it easier for recruiters to find your profile, and it will help you create a strong, consistent network of professional contacts. Setting up your LinkedIn page might seem annoying at the beginning, but you should consider it as a long-term investment. Obviously, LinkedIn can be of help to find a job by both searching and attracting connections, but the real first step to get a job is cracking interviews.

## Preparing for job interviews

Whether an interview happens physically or virtually, a few considerations are always valid. As a Xamarin jobseeker, you are applying for technical positions, so there will likely be more interviewers, typically, the manager of the team you should work with, a developer on the team, and a representative of the human resources department, at least in the first contacts. First, listen carefully to what they have to say about what the company does and what you should do as a developer working on their products. If possible, ask questions so that interviewers understand that you have real interest in the job.

## Technical interviews

Technical questions will usually be asked to understand whether your skills are okay to work on the company's products and not to put you in trouble or to understand how cool you are about software development. Remember that a company makes a huge investment when hiring people, so they must make sure that your technical skills and experience will benefit them. It might also happen, especially with very large businesses with complex hierarchies and regulations, that you will be asked to complete some technical exercises. Obviously, it is not possible to predict or discuss technical questions in detail; these depend on the interviewer, on the job position, and on the products that the company builds.

## Personal interviews

Managers and representatives of the human resources department might ask questions that are more related to understanding you as a person, whether you could integrate with the existing team, your expectations, ambitions, and so on. A question that is often asked is *where do you see yourself in 10 years?*. Though there is no effective answer to this question, it is always a good idea to reply in a way that gives the impression that you want to grow, both personally and technically, in the company. For example, if you apply for a position as a software developer, in 10 years, you could want to see yourself as the technical lead of a team of software developers. They perfectly know that there is no precise answer to this question, but they want to understand how you think about your growth inside the company. The first interview is generally about meeting each other, so never ask for salary information, unless the interviewers want to talk about this. It is obvious that gaining a salary is the motivation for working for every person on the planet, but the perception you should give is that you are interested in the job regardless of the salary. This is usually the topic of a second, or even third interview. So, you will always have an option to discuss the salary level and benefits; there is no need at all to do this during the first interview. At least for the first meeting to discuss the position, when possible, accept even if they are set up late in the evening or early in the morning. This might be intentionally done to see how flexible you are, and it does not mean it will happen in your regular working days. In addition, before an interview, try to understand if the company has a dress code. If so, dress in a way that makes them understand that you have no problem with their rules. Finally, a recurring yet fundamental recommendation is to always be honest about your skills, personal needs, and expectations. Not only interviewers appreciate and prefer this approach, but it is also important for them to trust you.

## Conclusion

In today’s world, finding a job is no longer something you can do the old-fashioned way. Modern jobs require modern ways of finding an occupation, especially for careers in the IT industry, and because recruiters work in a modern way, it is very important to share your knowledge on public channels, including LinkedIn, so that it will be easier for you to attract the interest of employers who are searching for skilled developers. But once you have gained your job, you need to stay successful in your role so that you can maintain your position and grow professionally. This is the topic of the next chapter.

## Points to remember

- Your resume should be concise, and it should highlight the skills that are required for the position you are applying for.
- At interviews, always be honest. A job is a long-term investment for you and the company that hires you, so trust is fundamental.
- Use LinkedIn to search for jobs based on your skills and knowledge.
- Publish contents, write articles, and join a community when possible, to build an online reputation.
- Attracting a job is the best option for you, so share on LinkedIn what you do online.

# CHAPTER 15

# Succeeding as a Mobile App Developer

## Introduction

Gaining technical knowledge and finding a job are the two major things for starting a career in the world of mobile app development, but you will want to keep your role over time and possibly evolve to new, higher positions. Though this can be personal, all developers can do a few things to improve themselves and be more successful in the software industry. This chapter describes some best practices that will help you not only maintain your role but also help you make a difference in the daily work of your team.

## Structure

In this chapter, we will cover the following topics:

- Developing your passion and curiosity
- Learning to be a team player
- Staying up to date with development technologies

## Objectives

By completing this chapter, you will learn the importance of keeping your passion and curiosity at high levels, how to work in team, and how to stay up to date with the relevant technologies you will use in your daily work.

**All the topics discussed in this chapter should be considered as suggestions, not rules. In addition, they should be applied according to your work/life balance and availability. The best suggestion is to not overcharge yourself and always ensure time to relax.**

## Developing your passion and curiosity

The more you work on projects, the more you learn. However, your daily work can sometimes be repetitive. For example, you might work on the same project for years, and even if you work on adding new features, you might stay on the same topics for a while. This should certainly not be a problem but keeping your passion and curiosity high and looking at something else are very important things.

## Experiment on custom projects

One of the better ways to nurture your passion for writing code is still writing code. Take some time to create simple apps outside of your regular job and outside of business requirements. This should not be seen as a second job or as another commitment in life, of course. You can dedicate the amount of time you want and whenever you want, but it is important to focus on something that makes you feel free to experiment without the pressure of your daily work, with the technology you have studied and that you use every day. Set a goal, for example, the type of application you would like to build according to your time and its features, and then start. You can do this even if you do not know in advance how to implement the features you desire. This can be a very good exercise to improve your knowledge by studying the development platform, and the more obstacles you overcome, the more you will feel motivated and passionate. As an implication, experimenting will give you knowledge that you will also be able to reuse in your daily work. This will make you smarter and faster in making technical decisions. With regard to working on custom projects, a good exercise is to use GitHub (**https://github.com**) as a remote repository for your code. The reason is that other developers can see your code, give suggestions on how to improve it, and raise problems; in other words, they can interact and share ideas, which is of great value. So, feel free to experiment. It is a great way to keep doing what you like with no pressure, and it will really help you nurture your passion for coding.

## Experimenting with third-party components

Several software companies produce libraries and components that target different development platforms and help save time in implementing custom features. For example, if you need to implement charts in your

Xamarin.Forms project, you will likely want to use a third-party component rather than writing your own views and logic. There are costs to choosing licenses and subscriptions, but the time and effort to build components on your own would probably cost much more, so it is something to consider. Consequently, many companies use third-party components produced by others. If you find some time to experiment with such components, you will also be able to make more appropriate technical suggestions or decisions in your daily work. The most important library vendors that target Xamarin.Forms are listed here:

- **Infragistics** (**https://www.infragistics.com**): They provide a very powerful suite of data-oriented views for Xamarin.Forms and native platforms. This suite lacks general purpose views, but it is one of the most powerful and appreciated. It is possible to use a free trial before purchasing a license.
- **Progress, formerly Telerik** (**https://www.telerik.com**): They offer dozens of views for Xamarin.Forms and native platforms. They provide a free trial that you can use before purchasing a license.
- **Syncfusion** (**https://www.syncfusion.com**): They not only offer one of the most popular and efficient suites of components and libraries for Xamarin.Forms and native platforms, but they also provide a free community license, which allows for building commercial products within certain conditions. Refer to their website for more details.
- **DevExpress** (**https://www.devexpress.com**): They provide a series of both paid and free views for Xamarin.Forms. Free components are all designed to work with data, but they work very well, and they are certainly worth a try.

Finding some time to experiment with third-party components is also useful when you have job interviews with teams that use such components. And even if you apply for a position in a team that uses components that are different from the ones you have experimented with, the fact that you had an approach to using third-party libraries will certainly be appreciated.

## Learning to be a team player

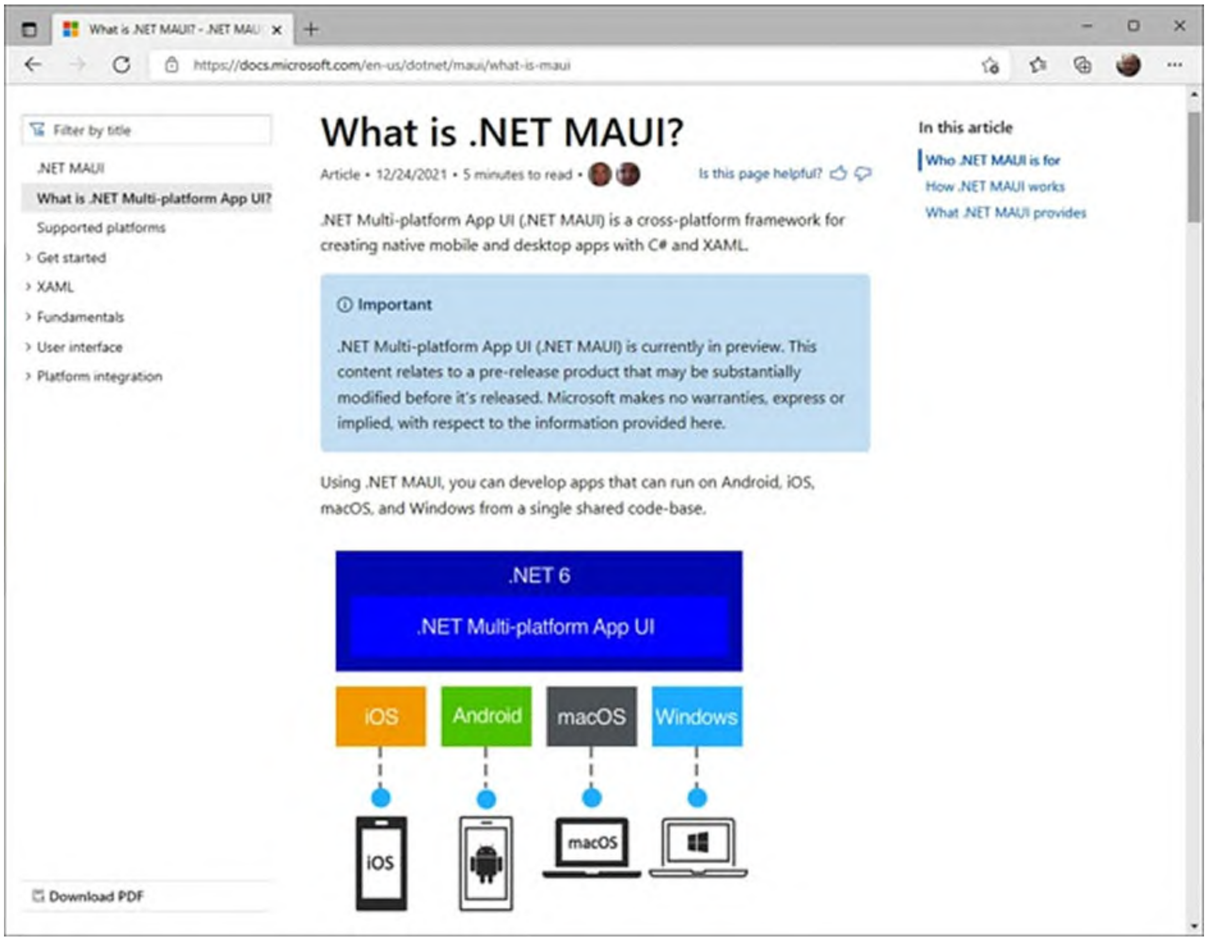
Building software is teamwork because several roles are involved, from graphic designers to developers, testers, translators, and managers such as product owners. When you join a company, you will likely work on a team. This is of great value for you because you can learn how processes work and how to interact with people who work on the same project but have different backgrounds. Learning the workflow of the team is fundamental; you will need some time to adapt to people and procedures, but then everything will become like a good habit. However, there is a big difference between being a member of a team and being a team player. Remember that, at least with the most modern work methodologies such as Agile, success or failures happen as a team; it is never the success or the fault of an individual. This means that everyone in the team, including you, should do their best to contribute to the project. If your work environment is smart enough, you can keep in mind the following suggestions:

- **Be proactive**: Do not wait for others to ask for ideas or feedback; if you see something that needs improvement in an app, share your thoughts and solutions with the other members.
- **Speak**: Share thoughts, doubts, and ideas with the others. In most cases, you will not make decisions on which features to implement, but you can give feedback that can fine-tune the impact they have on the end user.
- **Raise your hand**: Nobody is perfect, and no developer knows 100% of a development technology. So, you might have troubles, for example, when you need to work on a feature for the first time. If this happens, raise your hand; do not be ashamed of asking for help. Asking questions and asking for help is much better than writing code that you are doubtful about.
- **Be open to changes**: Not just technology, even business requirements and organizational needs evolve. The best approach you can keep is seeing such changes as value and not as problems.
- **Look at things with the eye of the team**: Even if a team is made of individuals, the worst thing that can happen is when every person thinks individually about their job instead of looking at the project outcome as a whole. Remember that it is the team that wins or fails, not individuals. Your customers or stakeholders will not care who has failed; they will consider a team’s failure and a team’s success.

## Staying up-to-date with development technologies

In *Chapter 1, The Importance of Mobile App Development*, you got suggestions on how to stay up to date with new releases of devices and operating systems from the major vendors in the market. Those suggestions remain valid, but they can be enriched with additional considerations now that you also have more technical skills. It is always important to take a look at new releases, especially about the development technologies you use daily. If you consider all the Microsoft technological stack discussed in the book, there are at least two technologies that you might want to analyze in further detail.

.**NET Multiplatform Application User Interface**, also known as .NET MAUI (**https://docs.microsoft.com/en-us/dotnet/maui/**), which is the new development platform from Microsoft that you will use to build mobile applications. Remember that all the skills you have learned in this book remain valid and fundamental with MAUI, but it is important that you read something about it. *Figure 15.1* shows how the welcome page of the official MAUI website looks:


What is .NET MAUI? - .NET MAUI

https://docs.microsoft.com/en-us/dotnet/maui/what-is-maui

Filter by title

.NET MAUI

What is .NET Multi-platform App UI?

Supported platforms

› Get started

› XAML

› Fundamentals

› User interface

› Platform integration

Download PDF

What is .NET MAUI?

Article • 12/24/2021 • 5 minutes to read •

Is this page helpful?

.NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.

Important

.NET Multi-platform App UI (.NET MAUI) is currently in preview. This content relates to a pre-release product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Using .NET MAUI, you can develop apps that can run on Android, iOS, macOS, and Windows from a single shared code-base.

In this article

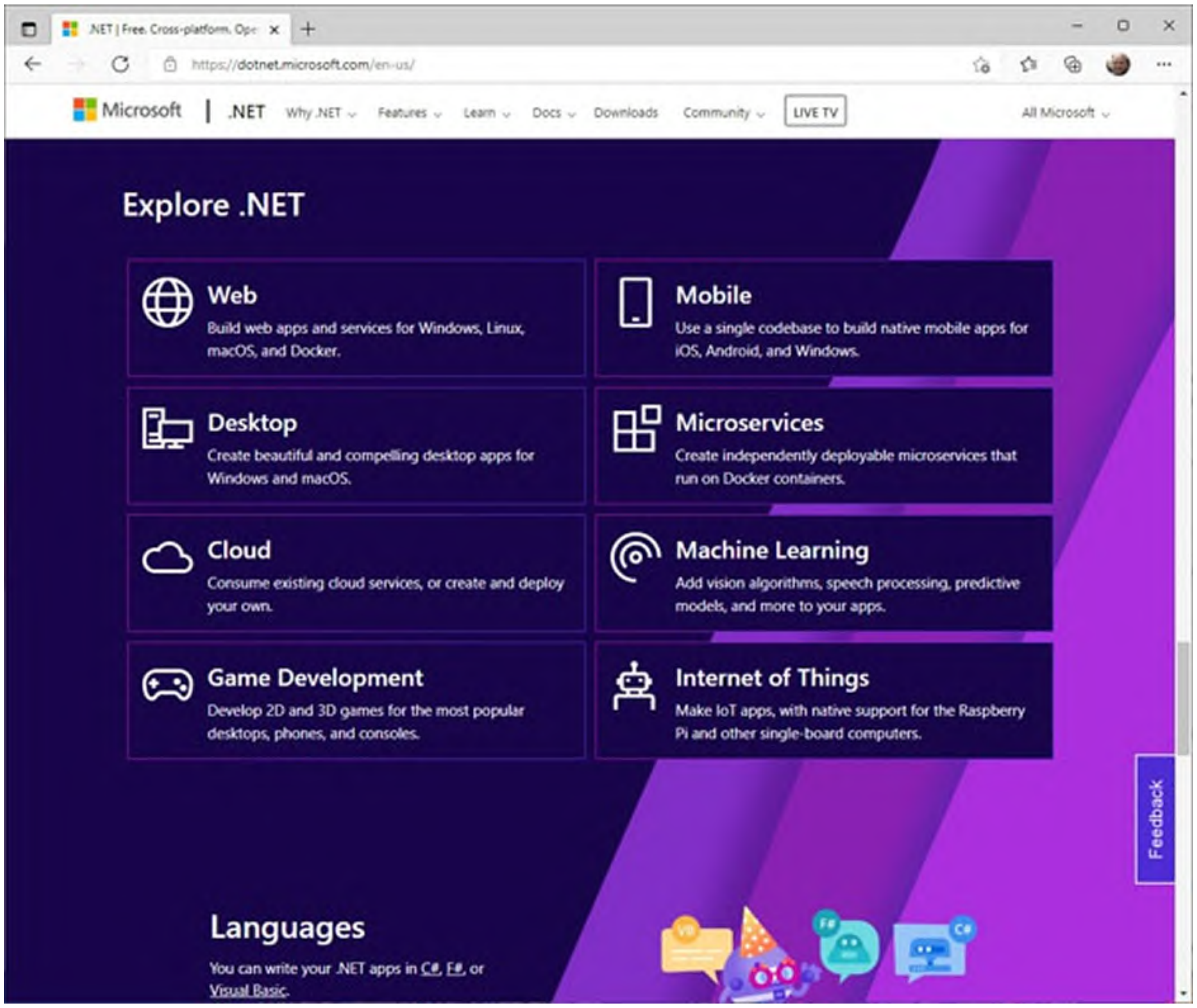
Who .NET MAUI is for

How .NET MAUI works

What .NET MAUI provides

***Figure 15.1:*** *The .NET MAUI welcome page*

.NET 6 (**https://dot.net**), released in November 2021, that finally reaches the goal of providing one API that targets all development areas. *Figure 15.2* shows how the welcome page of the official .NET site appears:



Explore .NET

**Web**
Build web apps and services for Windows, Linux, macOS, and Docker.

**Mobile**
Use a single codebase to build native mobile apps for iOS, Android, and Windows.

**Desktop**
Create beautiful and compelling desktop apps for Windows and macOS.

**Microservices**
Create independently deployable microservices that run on Docker containers.

**Cloud**
Consume existing cloud services, or create and deploy your own.

**Machine Learning**
Add vision algorithms, speech processing, predictive models, and more to your apps.

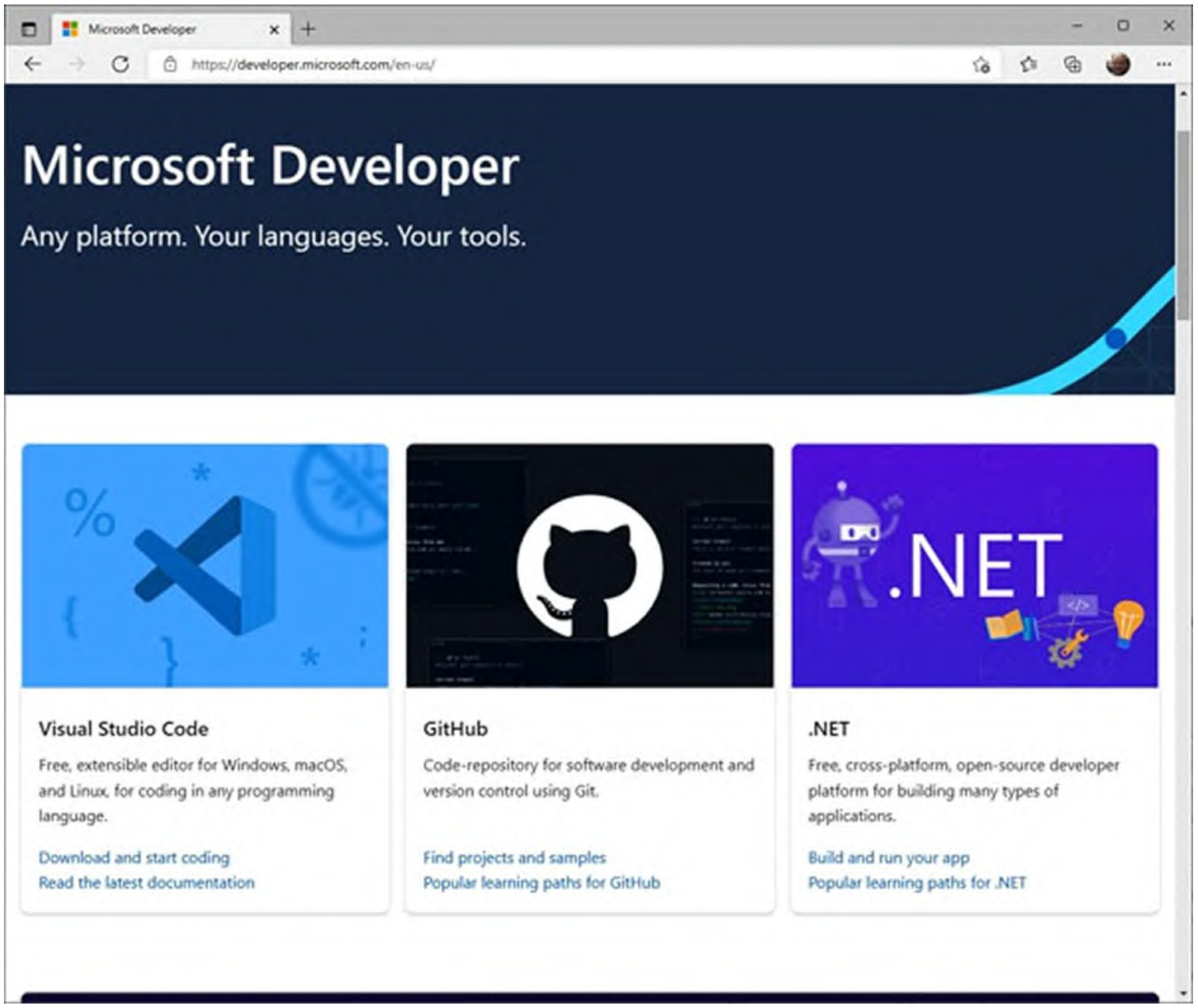
**Game Development**
Develop 2D and 3D games for the most popular desktops, phones, and consoles.

**Internet of Things**
Make IoT apps, with native support for the Raspberry Pi and other single-board computers.

Languages
You can write your .NET apps in C#, F#, or Visual Basic.

***Figure 15.2:*** *The .NET official website*

Generally, you can look at what Microsoft is building and releasing on their Microsoft Developer website (**https://developer.microsoft.com/**), which targets all supported development platforms. Here, you can find news, information, and shortcuts to the technologies you use every day as a Xamarin developer. *Figure 15.3* shows how the home page for the Microsoft Developer website looks:



Microsoft Developer

Any platform. Your languages. Your tools.

Visual Studio Code

Free, extensible editor for Windows, macOS, and Linux, for coding in any programming language.

Download and start coding
Read the latest documentation

GitHub

Code-repository for software development and version control using Git.

Find projects and samples
Popular learning paths for GitHub

.NET

Free, cross-platform, open-source developer platform for building many types of applications.

Build and run your app
Popular learning paths for .NET

***Figure 15.3:*** *The Microsoft Developer website*

It is really important to know what's coming next for several reasons. For example, if you need to start a new project, you can contribute to better decision-making about the technology to use. Another example is that you will be a point of reference to all the other colleagues who do not like to stay informed frequently. This will make a huge difference. This discussion can be connected to the suggestions about experimenting, discussed earlier in this chapter. Usually, Microsoft provides early builds (known as previews) of the products they are building, which you can use to start experimenting and look at what's new.

**Tip: Development teams are made of human beings, with their strengths and weaknesses. In your professional life, you will find people who like to stay up to date with technology and people who wait for others to get news about what's happening. Try to be in the first**

**category as you can benefit from it.**

In addition, this approach can be useful if you want to apply for new positions. The key point is not to have deep knowledge about each new technology, which would not be possible; it is to remember what a technology is about.

## Attending conferences and meetups

Developer communities across the world often organize free conferences, meetups, and webinars to discuss new technologies and development platforms or tools. The COVID-19 pandemic has boosted the availability of online conferences and meetups, so you can find those of your interest and attend sessions that discuss technologies that are closer to your daily work, from the comfort of your home. This is an incredible opportunity as until only a few years ago, developers could only attend paid, on-site conferences. It is important to take advantage of the good things that the internet offers, and this is one of them.

## Conclusion

Every developer can write code, but only a few developers will really make a difference. If you want to be one of them, you should nurture your passion and curiosity for technology and software development. There are several ways to do this, but the easiest and most attractive way is to experiment with new and early releases, third-party components, and your own projects outside of the daily work routine. Additionally, developing the way you work in a team will help keep motivation high, so it is recommended to learn to become a team player and not just work as a team member.

## Points to remember

- Developers are people with passion for technology. Keep this passion always on by never getting annoyed. Write your own code, try different things, and experiment.
- Successes and failures happen as a team. Help your team grow by becoming a team player. Be proactive, stay engaged, and raise your

hand when necessary.

- Staying up to date with technology is a must for a software developer. The more you know, the more you can do. You can also help in decision-making about technologies to use in a company's projects.

## Suggested readings

- How to be a great team player (**https://www.mindtools.com/pages/article/newTMM_53.htm**).
- How to make programming exciting and more fun (**https://www.freecodecamp.org/news/how-to-make-programming-more-exciting-and-funnier/**).
- The Agile Coach (**https://www.atlassian.com/agile**).

# Index

## Symbols

.NET 6
  architecture 38, 39
  URL 430
.NET Core 37
  advantages 37
  architecture 38
  components 37
.NET Framework 32
  architecture 34
  assemblies 36
  Base Class Library (BCL) 36
  Common Language Runtime (CLR) 34
  compilers 35, 36
  for Windows 33
  locating, on disk 33, 34
  to .NET Core 37
.NET MAUI 21, 22
  URL 429
.NET primitive types 72
  value types, versus reference types 74, 75
  variables, consuming 73, 74
  variables, declaring 73

## A

abstract classes 99, 100
access modifiers, in C# 89
advanced C# programming 109
  asynchronous programming 112-114
  generics type 109, 110
  Language INtegrated Query (LINQ) 111, 112
  nullable type 109-111
advanced data binding
  Model-View-ViewModel pattern 278
advanced view customization, native API
  Android renderer, defining 384-386
  custom renderers 383, 384
  custom renderers, applying to views 387
  custom view, defining 384
  iOS renderer, defining 386, 387
Android
  URL 12

anonymous types 74
Apple
  mobile device updates, obtaining from 11, 12
  URL 11
Application class 320, 321
  global variables, defining 322
  themes, working with 321, 322
application events
  data forms restoring, real-world example 325
  data retrieving, real-world example 324
  data storing, real-world example 324
  handling 323
  OnResume 323
  OnSleep 323
  OnStart 323
  page events, responding to 325, 326
application lifecycle 319
  events 323
  sample project, creating 320
Application Programming Interface (API) 7
applications, with Visual Studio
  building 39, 40
  first C# project, creating 45-47
  Microsoft Account, signing in 40, 41
  projects, creating with C# 43-45
  solution 43
  Start window 42
  synchronized settings, Microsoft Account 41
apps
  Android packages, preparing for publication 145-147
  iOS packages, preparing for publication 147
  preparing, for publication 145
arrays, in C# 87
  jagged arrays 88
  multi-dimensional arrays 87, 88
  single-dimensional arrays 87
assembly, .NET 36
asynchronous programming 112-114
auto-implemented properties 91
Azure 332
  free subscription 332

## B

Base Class Library (BCL) 34-36
Battery class 402
bindable layouts 276-278
Bitbucket
  URL 422
Boolean and numerical values

incremental value selection, with Stepper 199
multiple choices, with RadioButton 194-196
on and off options, with switch 192, 193
RadioButton multiple groups, implementing 196, 197
selecting 192
Switch, implementing 192
user choices, with CheckBox 193, 194
value selection, with Slider 197, 198
brushes
LinearGradientBrush 301, 302
objects, coloring with 300, 301
RadialGradientBrush 302, 303
SolidColorBrush 301

## C

CarouselPage 219-221
CarouselView 270
implementing 271, 272
cascading style sheets (CSS) 172
code debugging 60, 61
breakpoints, adding 63, 64
data tips 64, 65
runtime errors, debugging 65-67
sample project, preparing 61-63
code editing features, Visual Studio IDE
code editor options, changing 57, 58
code editor, zooming 56
IntelliSense 56, 57
syntax colorization 55, 56
working with 55
CollectionView 265-268
command 283
Common Language Runtime (CLR) 34, 35
Common Type System 70
compiler, .NET 35
components, .NET Core
CoreCLR 37, 38
conditional code blocks, C# 85
if conditional code block 85
switch conditional code block 86
ContentPage 212, 213
conversion operators, C# 77
converting types, with Convert class 78
direct type conversion 77, 78
CoreCLR 37
C# programming 69
advanced C# programming 109
arrays 87
common data operators 75

conditional code blocks 85
data types 70
iterations 82
loops 84
object-oriented programming (OOP) 88
cross-platform access, to native API
battery status, checking 401-403
contents, opening 405
default apps, opening 405, 406
emails, sending 403, 404
network connection, checking 399-401
secure settings, storing 407
secure settings, with app version tracking 408, 409
SMS messages, sending 404, 405
user preferences, storing 406
web browser, opening 405
cross-platform approach 7, 8
application performance issues, demystifying 10
application size and performance 9
cross-platform code, limitations 9
native applications 9, 10
cross-platform development 8
technologies 8, 9
cross-platform projects, with Xamarin.Forms 139
apps, debugging 144
apps, running 144
App.xaml 142, 143
creating 139, 140
MainPage.xaml 142, 143
structure 141
XAML 142, 143
CSS styles
creating 174
implementing, in C# 174, 175
custom renderers 388, 389
applying, to view 387, 388

## D

data binding 246
advanced data binding 278
binding context, assigning 252, 253
different types, binding with value converters 261-264
implementing 247-249
property change notifications, with INotifyPropertyChanged 249-252
to user interface 246
data collections
data, displaying with ListView 254-259
DataTemplate, as resource 259-261
displaying efficiently 265

item indicators, displaying with IndicatorView 273, 274
items, selecting with Picker 274-276
lists, displaying with CollectionView 265-270
lists, scrolling with CarouselView 270-273
working with 253, 254
data operators, C# 75
arithmetic operators 76, 77
conversion operators 77
equals operator 75, 76
logical operators 79
not equals operator 75, 76
strings, working with 79, 80
data types, C# 70
Common Type System 70, 71
.NET primitive types 72
reference types 71
value types 71
DatePicker
dates, selecting with 188, 189
DependencyService class 397
developer mode 26
development tools
Android devices, configuring for developer mode 26, 27
configuring 22
installing 22
macOS configuration 27, 28
Microsoft Visual Studio 24
on Android device 24
on iOS device 24
system requirements, for Xamarin 23
device class, native API
conditional XAML 382, 383
device-based content orientation 382
thread-safe code, running 381
timers, working with 380, 381
working with 379, 380
do loop 84

## E

Error List tool window 51
Events page
URL 14
eXtensible Application Markup Language (XAML) 18
Extensible Markup Language (XML) 43

## F

FlyoutPage 213-217

## G

GitHub
  URL 422
Google
  mobile device updates, obtaining from 12
Google Developers
  URL 13
Grid 159, 161
  proportions, for rows and columns 161
  spacing 161
  spans, for rows and columns 161

## H

HTML content
  displaying 191, 192

## I

if conditional code block 85
images, Xamarin.Forms.View
  Aspect, handling 204
  displaying 202-204
  managing 205
implicit styling 246
IndicatorView 273
  implementing 273, 274
inheritance 96
IntelliSense 56, 57
  auto-completion options 57
interactivity, Xamarin.Forms.View
  adding 205
  alerts, displaying 207
  alerts, handling 207
  GestureRecognizers, implementing 206
interfaces
  defining 97, 98
  implementing 98, 99
IsRefreshing 288
iterations 82, 83

## J

jagged arrays 88
JavaScript Object Notation (JSON) 334-337
Java Software Development Kit (SDK) 20
job interviews
  personal interviews 423

preparing for 422
technical interviews 423
jobs
finding, LinkedIn used 415-418
searching 418, 419
jobs, attracting with LinkedIn 419
articles, writing 421
code sharing 422
contents, sharing from others 419, 420
content writing 421
knowledge sharing 421
LinkedIn considerations 422
online communities, joining 421
right contacts, connecting 419
just-in-time (JIT) compiler 36

## L

Language INtegrated Query (LINQ) 111, 112
layouts 150, 151
alignment options 151, 152
.NET objects hierarchy 154
spacing options 152
visual tree 153
LinearGradientBrush 301, 302
loops, C# 84
do loop 84
while loop 84, 85

## M

MacInCloud
URL 22
managed code 35
managed environment 35
managed heap 71
master-details user interface 213
messages
broadcasting, with MVVM 327-329
sending, through app 326, 327
MessagingCenter
tips and tricks 329, 330
method 91
Microsoft
URL 13
Microsoft Developer
URL 14
Microsoft Developer website
URL 430
Microsoft, in mobile app ecosystem 18, 19

improved productivity, with Microsoft tools and services 19
Microsoft investments, on Xamarin 19
Microsoft Visual Studio 24
configuring 24, 25
individual components, checking for 26
installing 24
workloads, selecting 25
mobile app developer
conferences and meetups, attending 432
curiosity, developing 426
custom projects, experimenting 426
development technologies, updating 428-432
passion, developing 426
team player 427, 428
third-party components, experimenting 427
mobile app development
career opportunities 5
money making 5, 6
significance 2
mobile apps, for companies
examples 4, 5
significance 4
mobile device 2, 3
for consumers 3, 4
history 3
updates, from Apple 11, 12
updates, from Google 12, 13
updates, from Microsoft 13, 14
updating 10, 11
mobile system
working 6, 7
Model-View-ViewModel pattern 278
actions, defining with commands 283, 284
business logic, implementing 281
data, exposing 281-283
data model, defining 279, 281
frameworks 287
RefreshView 287-289
user interface, defining 284-287
viewmodels 281
MonoDevelop 17
MonoDroid 17
mono project
URL 17
MonoTouch 17
multi-dimensional arrays 87
multimedia
local file, playing 316
MediaElement, implementing 313, 314
media file, controlling 314, 315

working with 312

## N

namespaces 100
types and members 100-102
native API
cross-platform, accessing 399
device class, working with 379, 380
device features, accessing 397-399
plugins 409, 410
sample project, preparing 378
native properties
code, testing 394
effects, declaring 389, 390
managing, with effects 389
platform-specific effects, implementing 390-393
native views
customizing, with platform-specifics 395-397
displaying 394, 395

## O

object-oriented programming (OOP), in C# 88
abstract classes 97
access modifiers 89
actions, running with methods 91-94
constructors and static classes 94-96
derived types, defining with inheritance 96, 97
enumerations, defining 104, 105
enumerations, using 105
error handling, implementing 106, 107
events, implementing 107-109
information, storing with fields and properties 90, 91
interfaces 97
reference types, defining with classes 89, 90
structure, assigning to variables 104
structures' members 104
structure visibility 104
types, accessing within namespaces 102, 103
types, organizing within namespaces 100-102
value types, defining with structures 103, 104
OnIdiom 383
OnPlatform 382
Output window 52, 53
overloading 93

## P

page navigation 221-223

custom titles, implementing 225
data sharing 223, 224
lifecycle, of page 225, 226
physical back button, intercepting 226, 227
pages 212
CarouselPage 219-221
ContentPage 212, 213
FlyoutPage 213-217
TabbedPage 217-219
Picker 274
P/Invoke 35
Postman
about 333
URL 333
project compilation
code, debugging 58-61
code, running 58, 59
code, testing 60
configurations 58, 59
debug 58
release 59
properties 88
Properties window 51, 52
property changed notification 249
provisioning profiles 135

## R

RadialGradientBrush 302, 303
reference types 71
RefreshView 287-289
REpresentational State Transfer (REST) approach 334
resources
defining 241, 242
implicit styling, implementing 246
style inheritance, implementing 245
styles, assigning 242-244
styles, defining 242
resume
preparing 414, 415

## S

search functionalities, Xamarin.Forms.View
implementing 200, 201
shapes 304
circles, drawing 304
custom shapes, drawing 308, 309
drawing 304
ellipses, drawing 304, 305

  lines, drawing 306, 307
  path and geometries 309-311
  polygons, drawing 308
  rectangles, drawing 305, 306
Shell 227, 228
  flyout menu, adding 229, 230
  flyout menu, implementing 231, 232
  interacting programmatically 235, 236
  search bar, implementing 232-235
  structure 228, 229
  styles, changing 236, 237
  tab bar, creating 231
  tab bar, implementing 231, 232
Simple Object Access Protocol (SOAP) 333
single-dimensional arrays 87
SolidColorBrush 301
Solution Explorer 48-51
SQLite 289
  data access layer, implementing 292-294
  data access layer, invoking 295, 296
  database path, obtaining 290, 291
  data model, implementing 291, 292
  local data, accessing with 289
  user interface, extending 297, 298
SQLite NuGet package
  installing 289, 290
stack 74
string interpolation 80
strings 79
  concatenating strings 80, 81
  empty string, checking for 80
  formatting 81, 82
  null string, checking for 80
  working with 80
style inheritance
  implementing 245
styles
  assigning 244
  defining 242, 243
swiping 219
switch conditional code block 86
symbols 60
Synchronized Settings, Microsoft Account
  Startup settings 41
  theme settings 41
syntax colorization 55, 56
system requirements, for Xamarin
  on macOS 23, 24
  on Windows 23

## T

TabbedPage 217-219
text, Xamarin.Forms.View
  complex text formatting, with FormattedString 185, 186
  custom fonts, implementing 184, 185
  displaying, with label 180, 181
  fonts, applying 183, 184
  fonts, managing 183, 184
  passwords, entering 183
  user input, accepting with entry and editor 182, 183
  working with 180
TimePicker
  time, selecting with 189, 190
tool windows, Visual Studio IDE
  Error List 51
  Output window 52, 53
  Properties window 51, 52
  Solution Explorer 48-51

## U

unmanaged code 35
user interaction
  implementing, with buttons 186-188
user interface, organizing with layouts 154
  AbsoluteLayout 162, 163
  ContentView 168-170
  ContentView, using 170, 171
  FlexLayout 156-158
  Frame 167, 168
  Grid 159, 161
  RelativeLayout 164, 165
  sample project, creating 154
  ScrollView 166
  ScrollView, controlling programmatically 166, 167
  StackLayout 155, 156
user interface, styling with CSS 172
  CSS, defining as XAML resource 172, 173
  CSS files, consuming in XAML 173, 174

## V

value types 71
var keyword 74
views 177
visual states, Xamarin.Forms.View 208-210
Visual Studio 2019 40
Visual Studio IDE
  application settings, editing 54, 55

basic code editing features 55
project properties, editing 53, 54
tool windows, working with 47, 48
working with 47
visual tree 153

## W

Web API 333
controllers, implementing 346-348
creating, in Visual Studio 337-340
data model, creating 342-345
data objects, creating 349
data objects, deleting 351, 352
data objects, updating 349-351
data, retrieving 348, 349
prerequisites 332
project structure 340-342
Web API services
API calls, making with Postman 354-357
calling, from C# 369-372
publishing 357-363
testing 352-354
Web API, with Xamarin.Forms
client application, testing 375
consuming 363, 364
data model, creating 364-366
user interface, designing 372-374
ViewModel, creating 366-369
web service 333
while loop 84, 85
Windows Communication Foundation (WCF) 333
Windows Compact Edition (CE) 3
Windows Presentation Foundation WPF) 36

## X

Xamarin 20
Xamarin.Android 20
Xamarin.Forms 20
Xamarin.iOS 20
Xamarin.Mac 20
Xamarin.Android projects 116
Android app, debugging 123-125
App Manifest, configuring 120-122
build options, configuring 125-127
creating 116, 117
debugging, configuring 125-127
structure 118, 119
user interface, drawing 119

user interface, handling 120
Xamarin, as app development framework 16
mobile app development, with Microsoft skills 16
Xamarin, as company 16, 17
mono project 17
transition, to Xamarin 17, 18
Xamarin Community Toolkit
installing 312, 313
Xamarin.Essentials 399
essentials API 409
Xamarin.Forms 20
advantages 21
layouts 150, 151
Xamarin.Forms.View
Boolean and numerical values, selecting 192
common properties 179
dates, selecting with DatePicker 188, 189
HTML content, displaying 191, 192
images, displaying 202-204
interactivity, adding 205
long-running tasks, handling 201, 202
sample project, creating 178
search functionalities, implementing 200, 201
text, working with 180
time, selecting with TimePicker 189, 190
user interaction, with buttons 186-188
visual states 208-210
Xamarin.iOS projects 127
App Package options, configuring 136-138
creating 127-129
developer accounts 135, 136
iOS app, debugging 133-135
provisioning profiles 135, 136
structure 130, 131
user interface, handling 131, 132