.NET MAUI

SUCCINCTLY

BY ALESSANDRO DEL SOLE

SUCCINCTLY EBOOK SERIES

Syncfusion®

.NET MAUI Succinctly

Alessandro Del Sole

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 111
Morrisville, NC 27560
USA

ISBN: 978-1-64200-237-9

Technical Reviewer: James McCaffrey
Copy Editors: Courtney Wright and Jacqueline Bieringer, content producer, Syncfusion, Inc.
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, content team lead, Syncfusion, Inc.

Table of Contents

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

Let us know what you think

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

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

Please follow us on social media and help us spread the word about the *Succinctly* series!



About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and a former Microsoft MVP. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and ebooks on programming with Visual Studio, including *.NET MAUI Succinctly*, *Visual Basic 2015 Unleashed*, and *Xamarin.Forms Succinctly*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals. He has been a frequent speaker at Italian conferences, and he has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior software engineer for Fresenius Medical Care, focusing on building mobile apps with .NET MAUI and Xamarin in the healthcare market. You can follow him on LinkedIn and support him with a coffee.

Introduction

This book is about the .NET Multi-platform App UI (.NET MAUI) development technology. It will be discussed in a way that allows both absolute beginners (who have at least a basic knowledge of C#) and expert developers to take advantage of this platform to build apps for Android, iOS, macOS, Windows, and Tizen from one C# codebase.

Understanding what led Microsoft to develop .NET MAUI is important to get the most out of the new features and architecture. When Microsoft acquired Xamarin back in 2016, it made huge investments in providing developers with first-class tools to build cross-platform projects with C#. The various flavors of the Xamarin technologies, such as Xamarin.Android, Xamarin.iOS, and Xamarin.Forms, have been fully integrated with Microsoft Visual Studio and its powerful code editor, debugging tools, testing platform, and so on. Not limited to this, the evolution of Xamarin as a development technology has continuously improved the way developers can leverage cross-platform code to build apps for Android and iOS using their existing skills on the Microsoft stack.

Like any other development technologies, Xamarin has its strengths and weaknesses. To make things easier for developers, Microsoft developed .NET MAUI, a cross-platform technology built from the ground up. It allows developers to write apps for Android, iOS, macOS, Windows, and Tizen from one shared C# codebase, but with three major evolutions: it is part of .NET; it was made to be integrated in Visual Studio, whereas for Xamarin, the integration came later; and it is lighter and better performing.

If you have never worked with Xamarin, you will find a very comfortable development environment whose model is an individual project that contains shared code and resources.

If you have worked with Xamarin, especially Xamarin.Forms, you will be pleased to see that your existing skills will not need to change and that you will be immediately ready to write code. This is because .NET MAUI has been written with developers and their real-world applications in mind. Because .NET MAUI is based on a simplified architecture, you will notice some changes compared to Xamarin.Forms—especially in the project system and in how you manage shared resources—but this is all to offer a simplified experience.

With this book, you will learn the foundation of .NET MAUI and how to develop rich applications that target multiple systems, with highlights on the Visual Studio tooling and comparisons with Xamarin.Forms where appropriate.

Chapter 1 Getting Started with .NET MAUI

Before you start writing mobile apps with .NET MAUI, you first need to understand the state of mobile app development today, and how this technology fits into it. You also need to set up your development environment to be able to build, test, debug, and deploy your apps to Android, iOS, Mac, and Windows devices.

This chapter introduces .NET MAUI along with the tools and hardware you need for real-world development, and provides a high-level overview of the technology that prepares you for the next chapters.

Note: In this book, I will refer to the technology as .NET MAUI, or simply MAUI, interchangeably.

Introducing .NET MAUI

.NET MAUI stands for Multi-platform App UI, and it is a Microsoft technology for the development of native apps for iOS, Android, macOS, Tizen, and Windows from one shared C# codebase.

.NET MAUI is part of .NET and can be considered the successor of Xamarin.Forms. However, it is not just an evolution of Xamarin.Forms. .NET MAUI is a completely new technology written from scratch, providing better performance, code organization, and ease of development. If you have already worked with Xamarin.Forms, you will see how .NET MAUI fully allows you to reuse your existing skills and experience with only a few upgrades.

The last chapter of this book provides input on migrating Xamarin.Forms projects to MAUI, but while reading you will understand on your own that Microsoft has built the new technology keeping in mind that migration should be affordable.

The main goal of .NET MAUI, like it was for Xamarin.Forms, is making it easier for .NET developers to build native apps for Android, iOS, macOS, Tizen, and Windows by reusing their existing C# skills and knowledge. The reason behind this goal is simple: building apps for Android requires you to know Java and Android Studio or Eclipse; building apps for iOS requires you to know Objective-C or Swift and Xcode; building apps for Windows requires you to know C# and Visual Studio.

Regardless of whether you are an experienced or beginner .NET developer, getting to know all the possible platforms, languages, and development environments is extremely difficult, and costs are extremely high.

From a development point of view, .NET MAUI wraps native APIs into managed objects. The biggest benefit of MAUI is that you write code once, and it will run on all the supported platforms at no additional cost. Accessing native, platform-specific objects and APIs is possible in several ways, all discussed in the next chapters, but it requires some extra work.

Additionally, MAUI integrates with the Visual Studio IDE on Windows, but being part of .NET means it's possible to develop applications on different systems.

This book focuses on .NET MAUI with Visual Studio 2022 on Windows. Due to the announcement of the retirement of Visual Studio for Mac, you could consider using Visual Studio Code with the .NET MAUI extension if you do not work on Windows. However, different tools and systems will not be covered in this book. The technical concepts for .NET MAUI certainly apply regardless of the IDE you use, though.

Supported platforms

Out of the box, .NET MAUI allows creating apps for Android, iOS, Tizen, macOS, and Windows from a single C# codebase. There are important considerations that need to be made at this point:

- .NET MAUI can target macOS via Mac Catalyst. This is bridge software that makes it possible to bring apps for the iPad to macOS.
- In addition to Mac Catalyst, your Mac needs the .NET MAUI runtime components.
- Tizen is an operating system backed by Samsung and based on Linux, which has the purpose of empowering different kinds of devices.
- For Windows, .NET MAUI targets this system via Windows UI Library 3 (WinUI in short), the most recent framework for building applications for Windows 10 and 11. So, references to Windows in this book actually relate to WinUI.

In the next section you will learn how to install and configure all the necessary tools to work with .NET MAUI.

Setting up the development environment

In order to build native mobile apps with .NET MAUI, you need either Windows 10 or Windows 11 as your operating system and Microsoft Visual Studio 2022 as your development environment.

> ***Note: The minimum required version of Visual Studio 2022 for MAUI development is 17.3.0, but it is recommended that you keep your development environment up to date with the latest version.***

You can download and install the Visual Studio 2022 Community edition for free and get all the necessary tools for MAUI development.

When you start the installation, you will need to select the **.NET Multi-platform App UI development** workload in the Visual Studio Installer (see Figure 1).

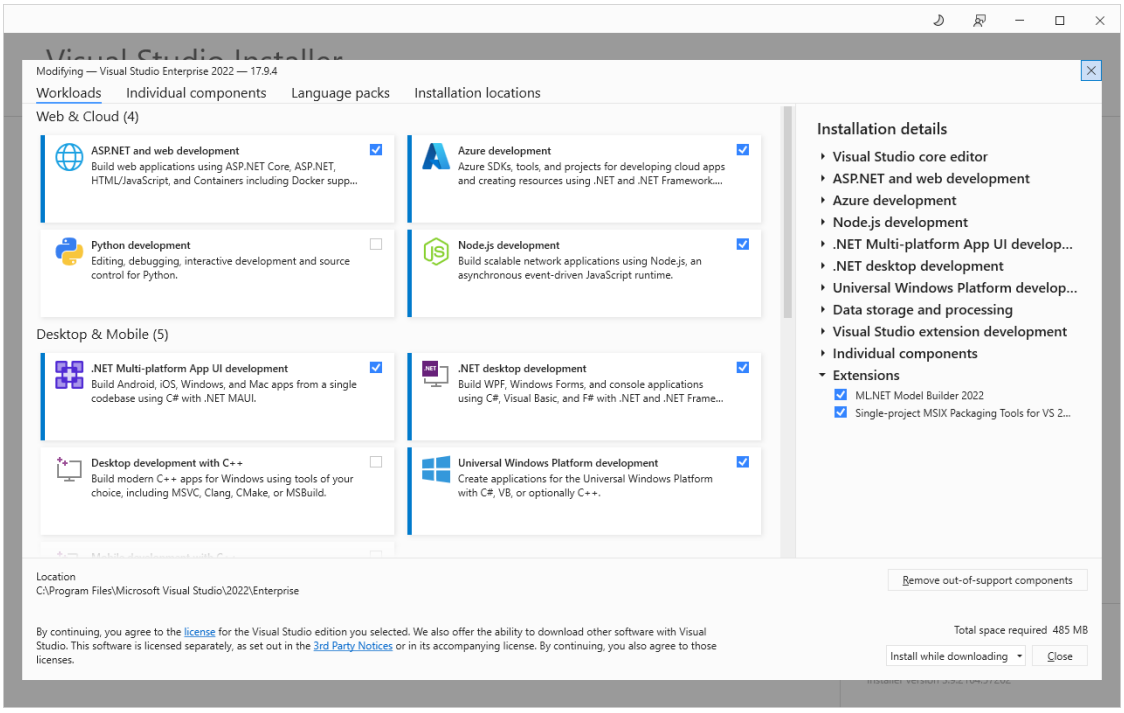

Figure 1: Installing .NET MAUI development tools

When you select this workload, the Visual Studio Installer will download and install all the necessary tools to build apps for Android, iOS, Tizen, and Windows. iOS requires additional configuration, described in the next section.

For Windows development, you will need additional tools and SDKs, which you can get by also selecting the **Universal Windows Platform development** workload. If you select the **Individual components** tab, you will be able to check if Android and Windows emulators have been selected or make a choice manually (see Figure 2).

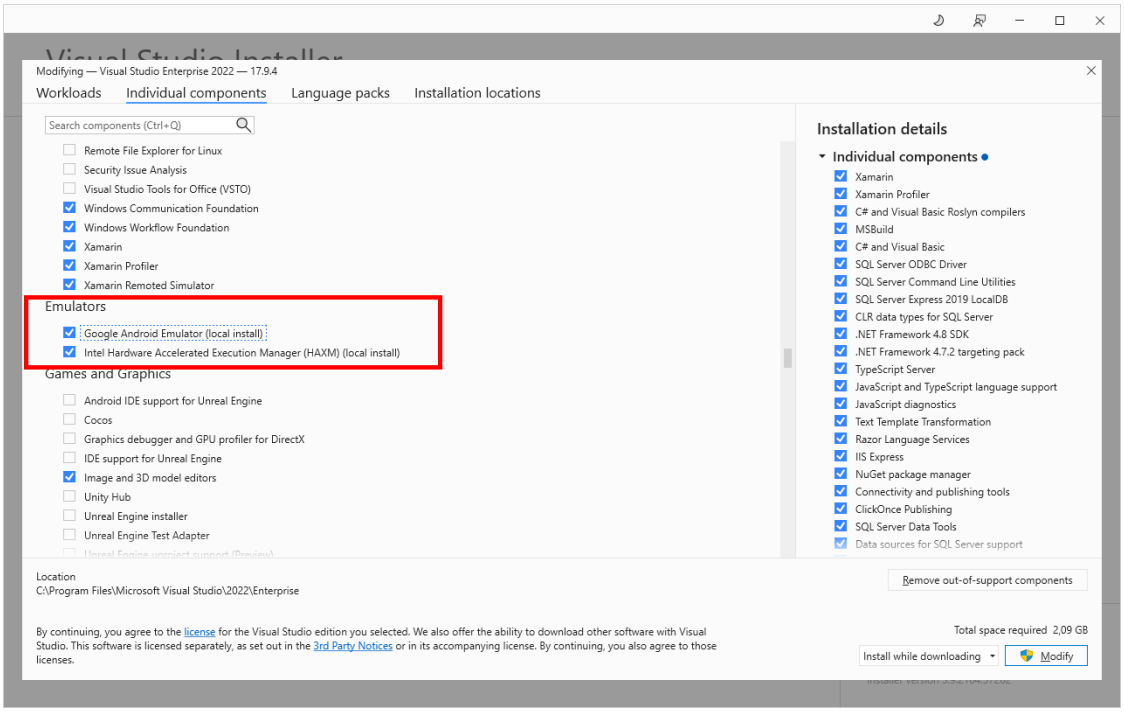

Figure 2: Selecting emulators

Whether you use Visual Studio 2022, Visual Studio for Mac, or both, I suggest you install the Google Android Emulator, which has an identical appearance and behavior on both systems.

Configuring a Mac

Apple's policies establish that a Mac computer is required to build apps for macOS, iOS, and tvOS. This is because only the Xcode development environment and the Apple SDKs are allowed to run the build process. A Mac is also needed for code signing, setting up profiles, and publishing an app to the App Store. You can use a local Mac in your network, which also allows you to debug and test apps on a physical device, or a remote Mac. In both cases, macOS must be configured with the following software requirements:

- macOS High Sierra (10.13) or later.
- Xcode and Apple SDKs, which you get from the App Store for free. Xcode also includes Mac Catalyst, so you can target macOS computers easily.
- .NET 8 and the .NET MAUI workload. Visual Studio 2022 will take care of installing the required components on the remote Mac on your behalf.

To connect Visual Studio 2022 to a Mac and enable builds for iOS and Mac Catalyst, you do not need to have a .NET MAUI solution opened. You can simply select **Tools** > **iOS** > **Pair to Mac**. In the **Pair to Mac** dialog (see Figure 3), select the Mac computer you wish to connect to and, when prompted, enter the access credentials.

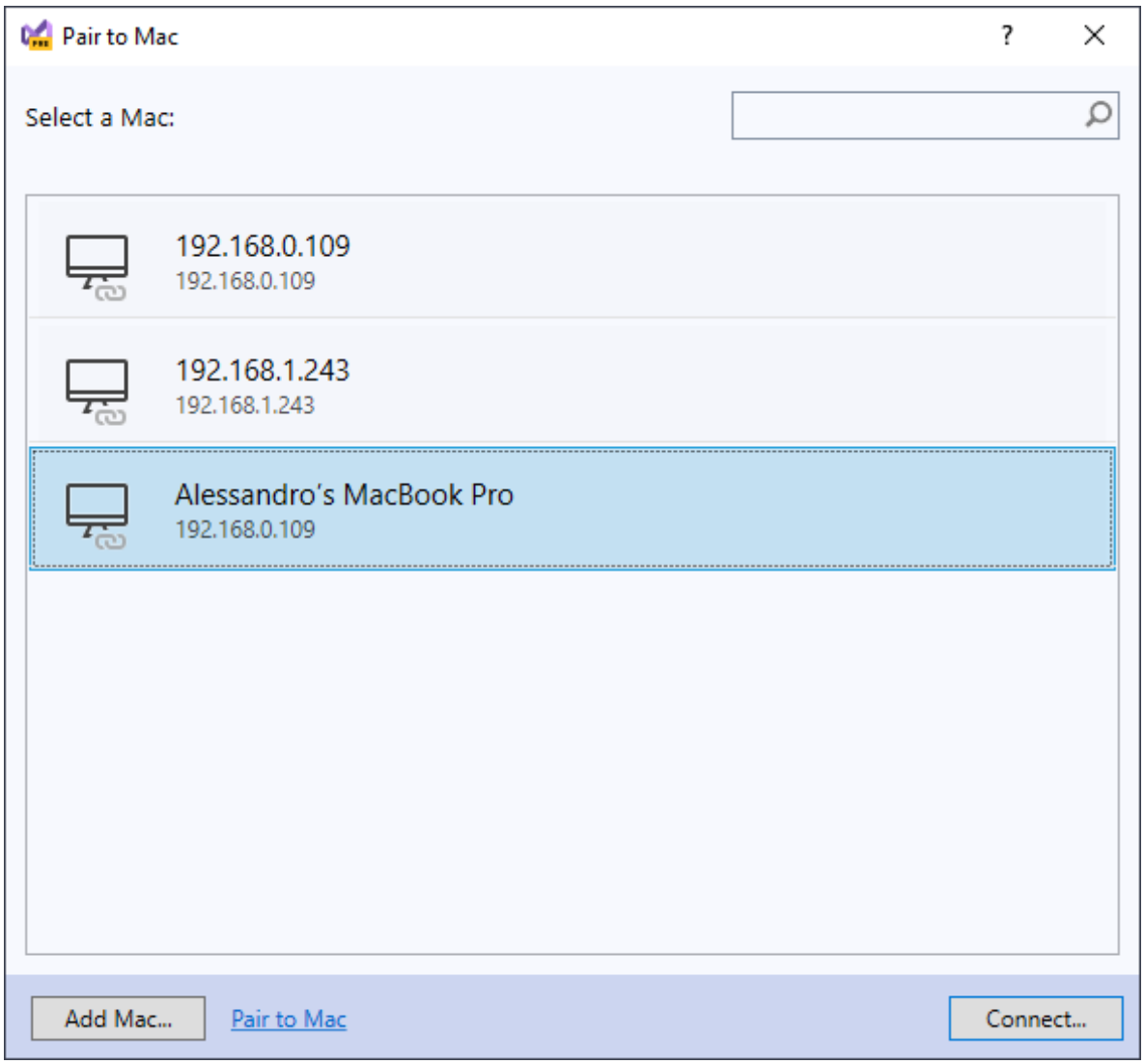

Figure 3: Pairing Visual Studio to a Mac

Visual Studio will first check if all the required components are installed and, if not, will try to install .NET 8 and the .NET MAUI workload remotely (see Figure 4).

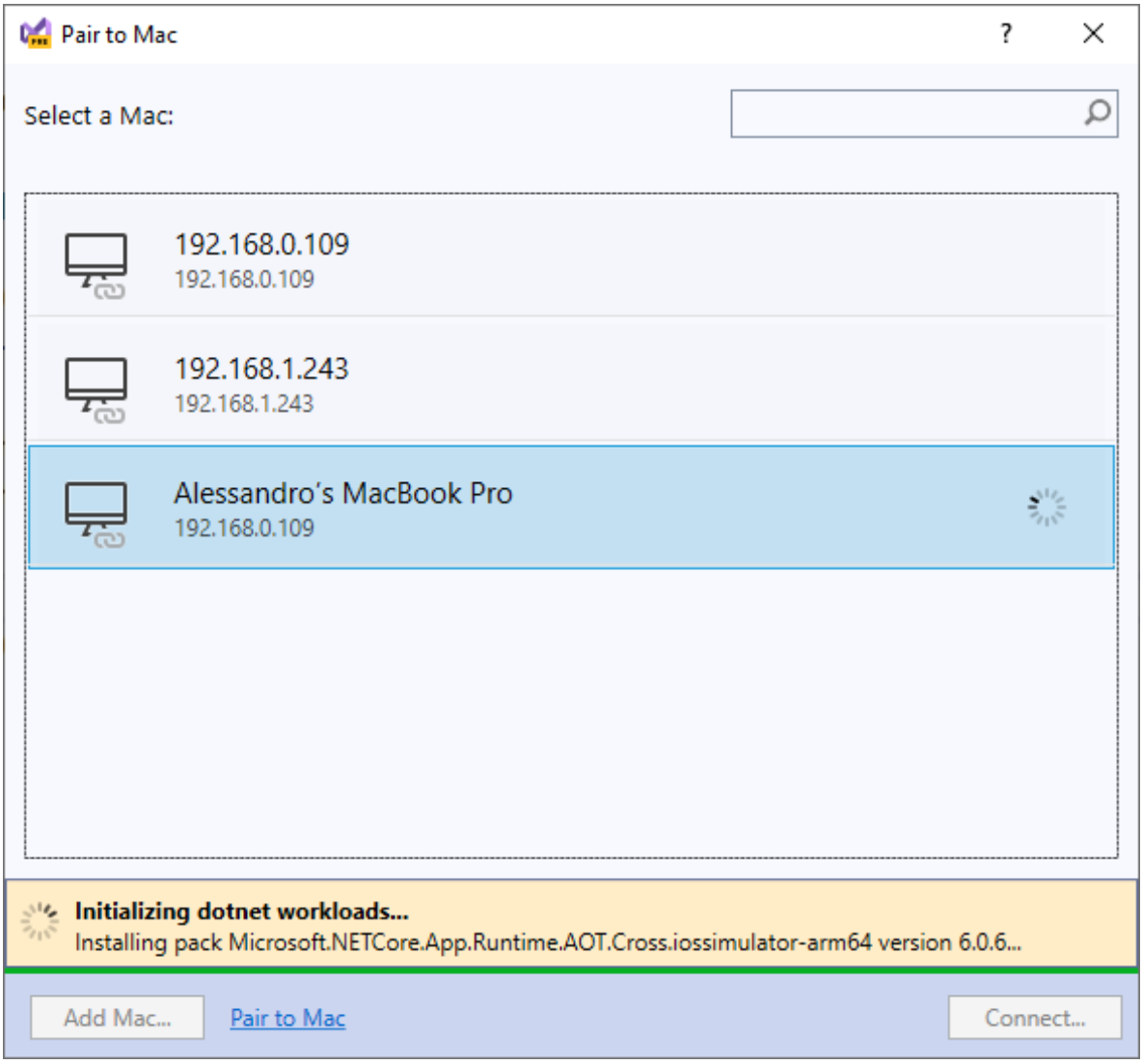

Figure 4: Visual Studio installs .NET MAUI components remotely

If Visual Studio 2022 encounters any problems while installing the .NET MAUI components to the remote Mac, you can solve the problem manually. On macOS, your first need to download and install the .NET 8 SDK. Second, you need to install the .NET MAUI workload by typing the following command inside an instance of the terminal:

```
> sudo dotnet workload install maui --source
https://api.nuget.org/v3/index.json
```

Every time you compile a .NET MAUI project, Visual Studio needs to be connected to a Mac to launch the Xcode compiler and SDKs. The official MAUI repository has a specific page that will help you configure a Mac. I recommend you read it carefully, especially because it explains how to configure profiles and certificates and how to use Xcode to perform preliminary configurations.

Creating .NET MAUI solutions

Assuming that you have installed and configured your development environment, the next step is opening Visual Studio to see how you create .NET MAUI solutions and what these solutions consist of. In the Start window, click **Create new project**. When the **Create a new project** dialog appears, enter MAUI in the search box to quickly filter the list of project templates. This will restrict the list of project templates to MAUI only.

The project templates of interest are **.NET MAUI App**, which is the first in the list (see Figure 5), **.NET MAUI Blazor App**, and **.NET MAUI Class Library**.

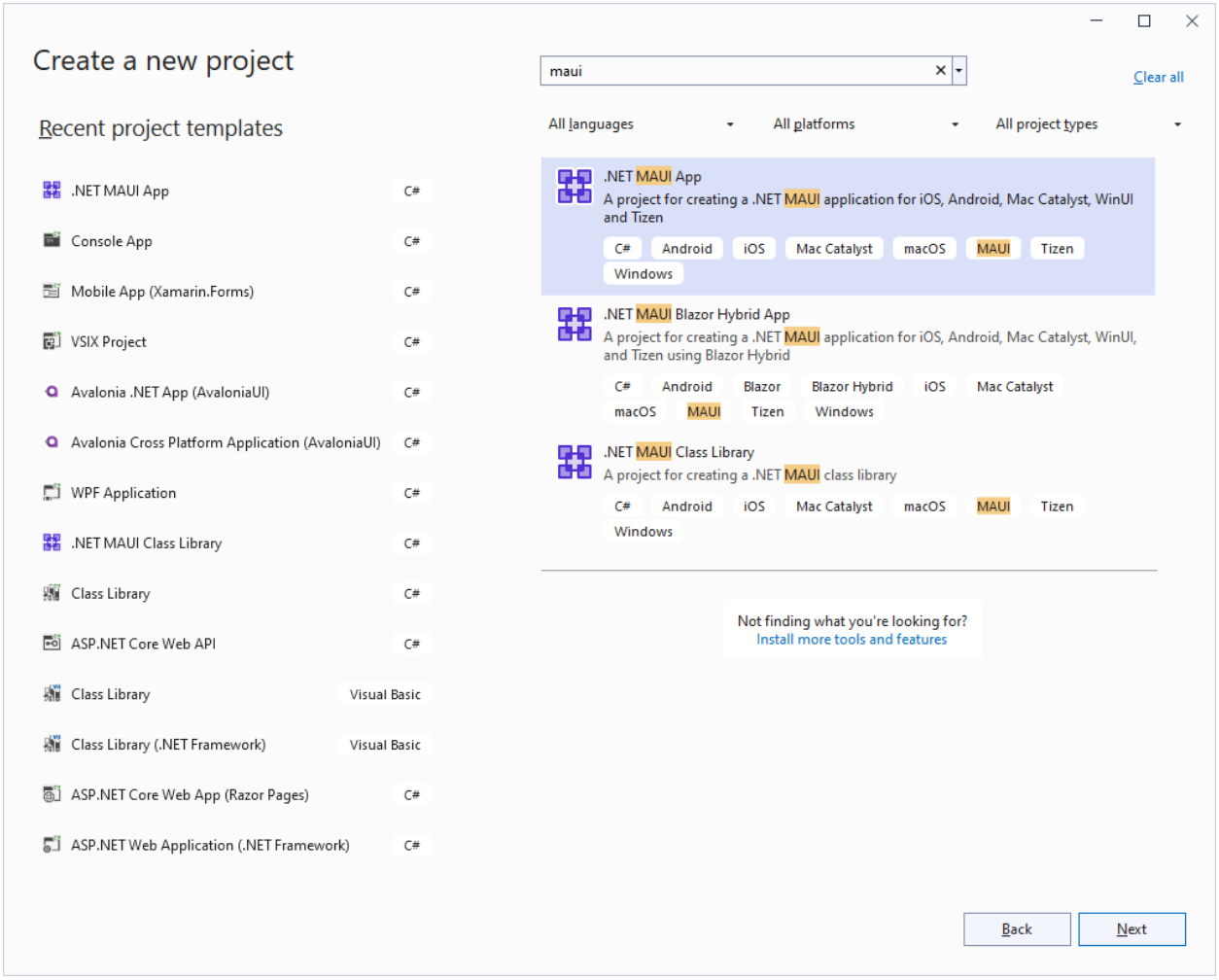

Figure 5: Project templates for .NET MAUI

Descriptions of the project templates follow:

- **.NET MAUI App**: This is the most commonly used project template, and the one you will use when following the explanations in this book. It allows for creating cross-platform projects targeting Android, iOS, Tizen, macOS, and Windows.

- **.NET MAUI Blazor Hybrid App**: This project template allows for embedding Blazor apps into .NET MAUI. This is related to a very specific scenario—hosting a Blazor app within a virtual browser inside a MAUI page—and will not be covered in this book.
- **.NET MAUI Class Library**: This project template can be used to create reusable class libraries on top of .NET 8 that can be shared across MAUI projects.

Select the **.NET MAUI App** template and click **Next**. In the next screen, Visual Studio will ask you to specify a project name and a location. Provide a name of your choice or leave the default name and then click **Next**.

The last step in the project creation is the selection of the target .NET version. Make sure you select .NET 8, as this is the most recent version at the time of writing.

Note: If you come from the Xamarin.Forms world, you know that Visual Studio asks you to specify which kind of UI structure to use in the project, such as flyout page, tabbed page, or blank. In .NET MAUI this is not the case because, as you will see shortly, the project template automatically generates the user interface based on the shell.

When the project is ready, you will notice one page called MainPage.xaml along with its code-behind file, and a new file called MauiProgram.cs. These are part of a new project system that represents crucial differences between MAUI and Xamarin.Forms, and that deserves a dedicated section.

Understanding the project system

The structure of a .NET MAUI solution is relatively simple, and it certainly is the biggest difference that experienced Xamarin.Forms developers will notice. The new project system is shown in Figure 6 through the Solution Explorer.

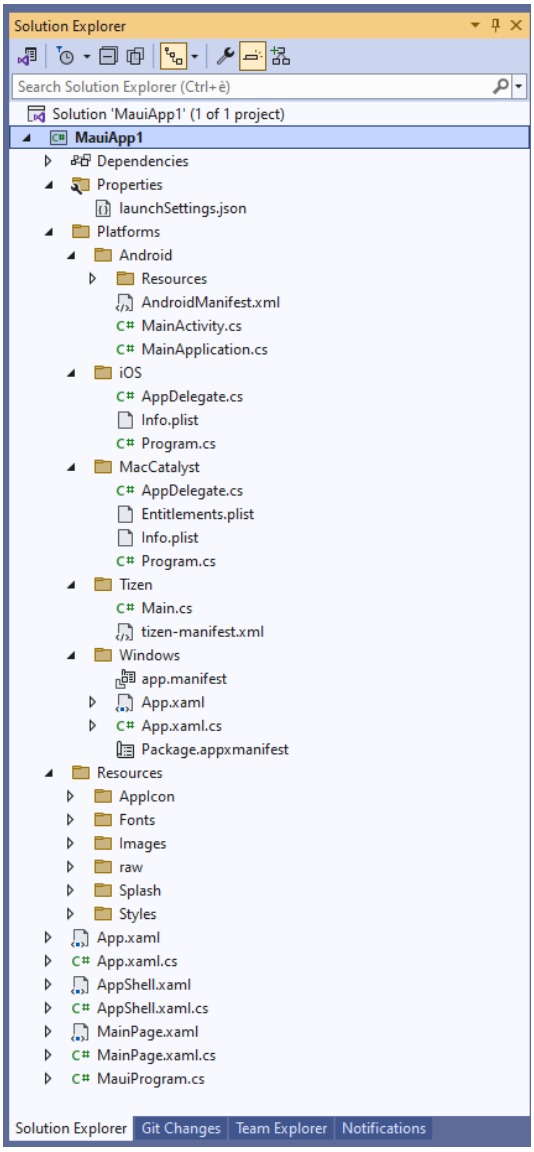

Figure 6: The project system in .NET MAUI

In the root folder, you find the following files:

- App.xaml with its C# code-behind file. This contains resources that are shared across the project and the startup objects.
- AppShell.xaml with its C# code-behind file. This file contains the definition of the Shell, an object discussed in Chapter 6 that provides common navigation infrastructure.
- MainPage.xaml with its C# code-behind. This is a default page that you can use to start developing your app.
- MauiProgram.cs. This is the main entry point of a MAUI project and is discussed at the end of this chapter.

The **Platforms** folder contains platform-specific subfolders that contain code and metadata that will only work on the related platform, and that .NET MAUI automatically resolves at runtime. The **Resources** folder is organized into the **AppIcon**, **Fonts**, **Images**, **Raw**, **Splash**, and **Styles** subfolders.

> ***Tip: There is a crucial difference from Xamarin.Forms here. In .NET MAUI, you add resources to one shared place. This saves time when adding resources and maintaining the project. In Xamarin.Forms, you had to add the same resources to all the projects in the solution, sometimes with the differentiations required by each platform.***

The AppIcon folder contains the application icon in .svg format. .NET MAUI supports the .svg file format without the need of third-party libraries. The Fonts folder contains font files. The Images folder contains image files required by your app. The Raw folder is the place where you add any raw resources that you want to include in your app (such as document files). The Splash folder contains the splash screen for the app, in .svg format. The Styles folder contains reusable XAML styles, which will be discussed in Chapter 7.

Before having a look at the platform-specific subfolders, it is a good idea to understand how to set project properties.

Configuring project properties

One of the advantages in .NET MAUI is that you can configure the output for each targeted platform in one place instead of dealing with multiple projects. To accomplish this, right-click the project name in Solution Explorer and select **Properties**. This will open the project properties user interface.

There are dozens of properties you can set, but only the most relevant are presented here. Under the **Application** tab, you can set the target platform for each system, as shown in Figure 7.

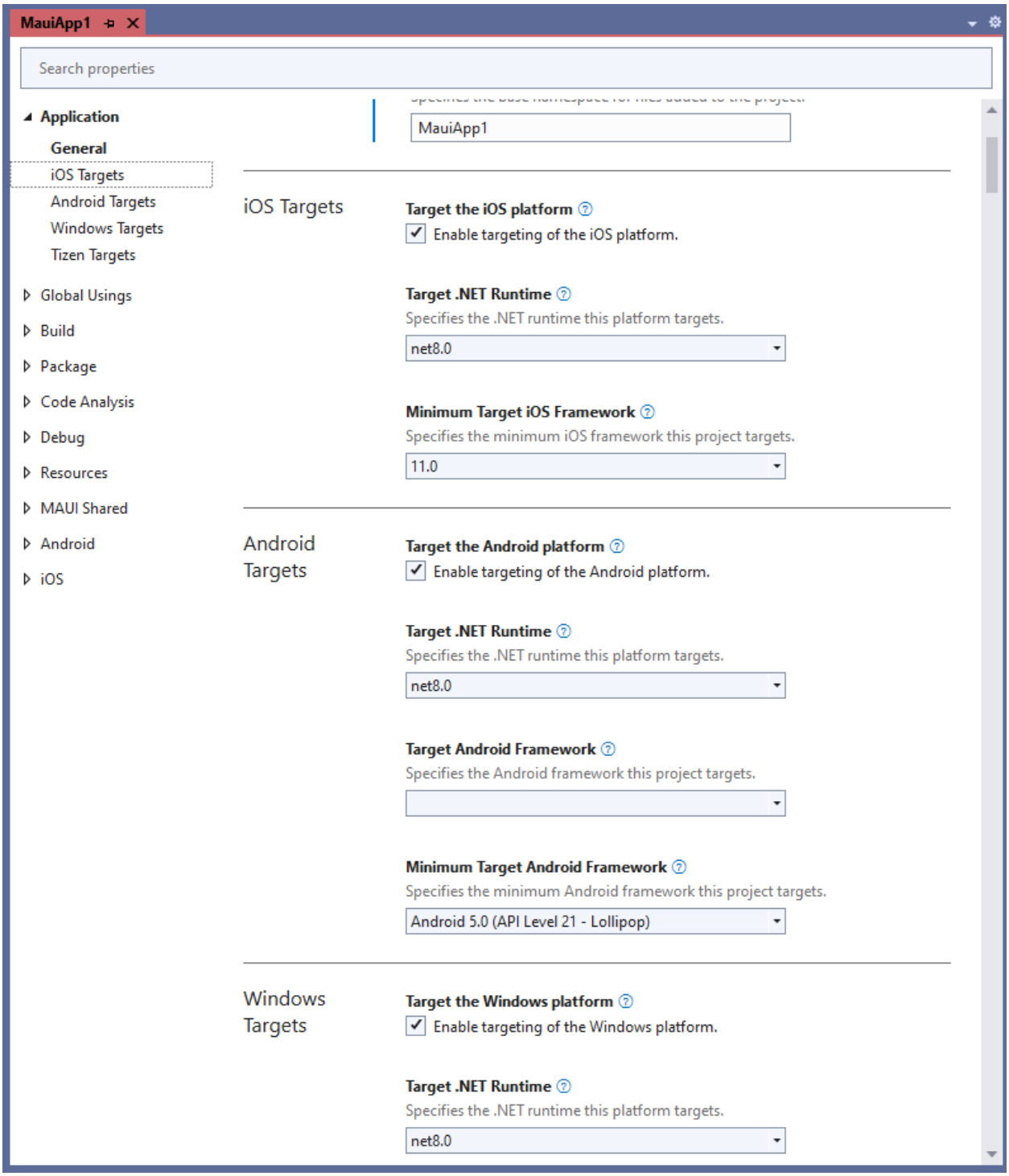

Figure 7: Assigning target frameworks

Here you can specify the target framework, which is based on .NET 8 for now, and the minimum OS version. For Android, you can also specify the minimum API level. API level 21 matches Android 5.0. In general, unless you have specific requirements, you can leave the default target frameworks unchanged. You can then move to the **MAUI Shared** tab (see Figure 8), where you can specify some app metadata like the name, version, and unique identifier for the stores.

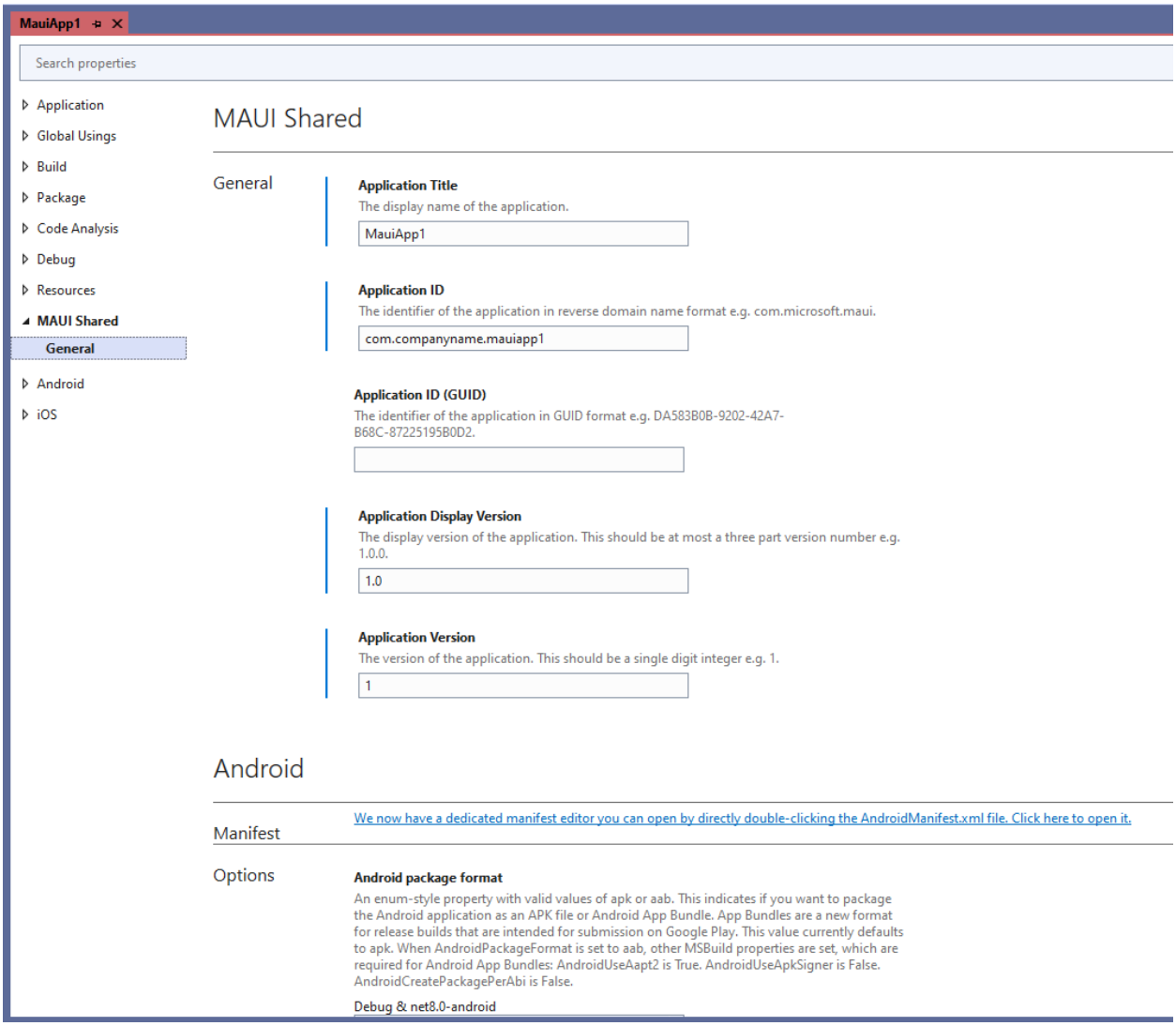

Figure 8: Assigning cross-platform project properties

In the **Application ID** text box, replace **`companyname`** with your name. By convention, application unique identifiers have the following form:

`com.companyname.projectname`

where **`com.`** is fixed. You should always follow this convention.

Configuring Android properties

At the bottom of the tab list on the left side, you will find the **Android** tab (see Figure 9).

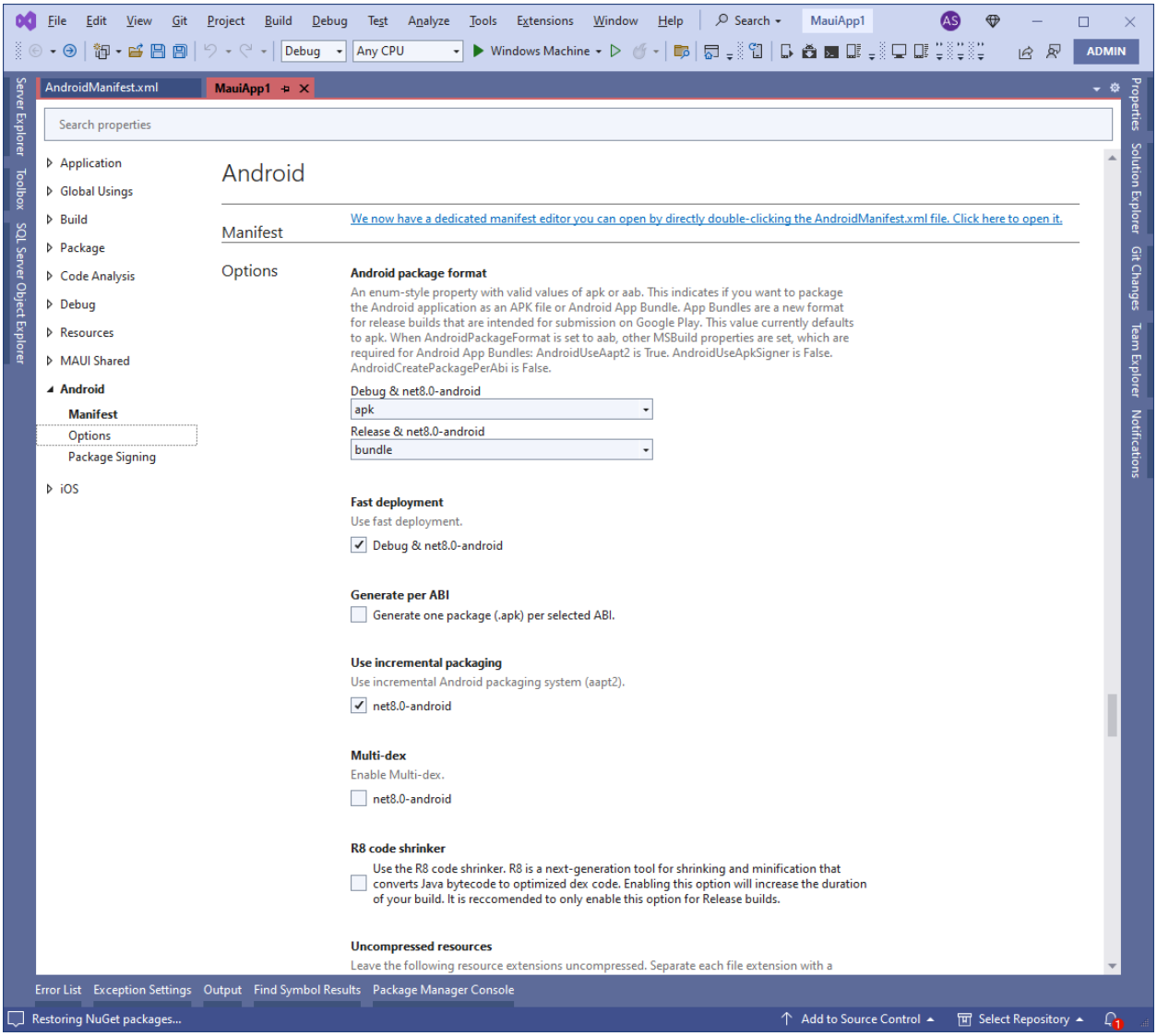

Figure 9: The Android options

It is composed of three configuration areas: Manifest, Options, and Package Signing. The manifest is represented by the **AndroidManifest.xml** file located in the **Platforms\Android** subfolder, and Visual Studio provides a dedicated editor, which you access by clicking the hyperlink that you see at the top of Figure 9. The manifest editor looks like the image in Figure 10.

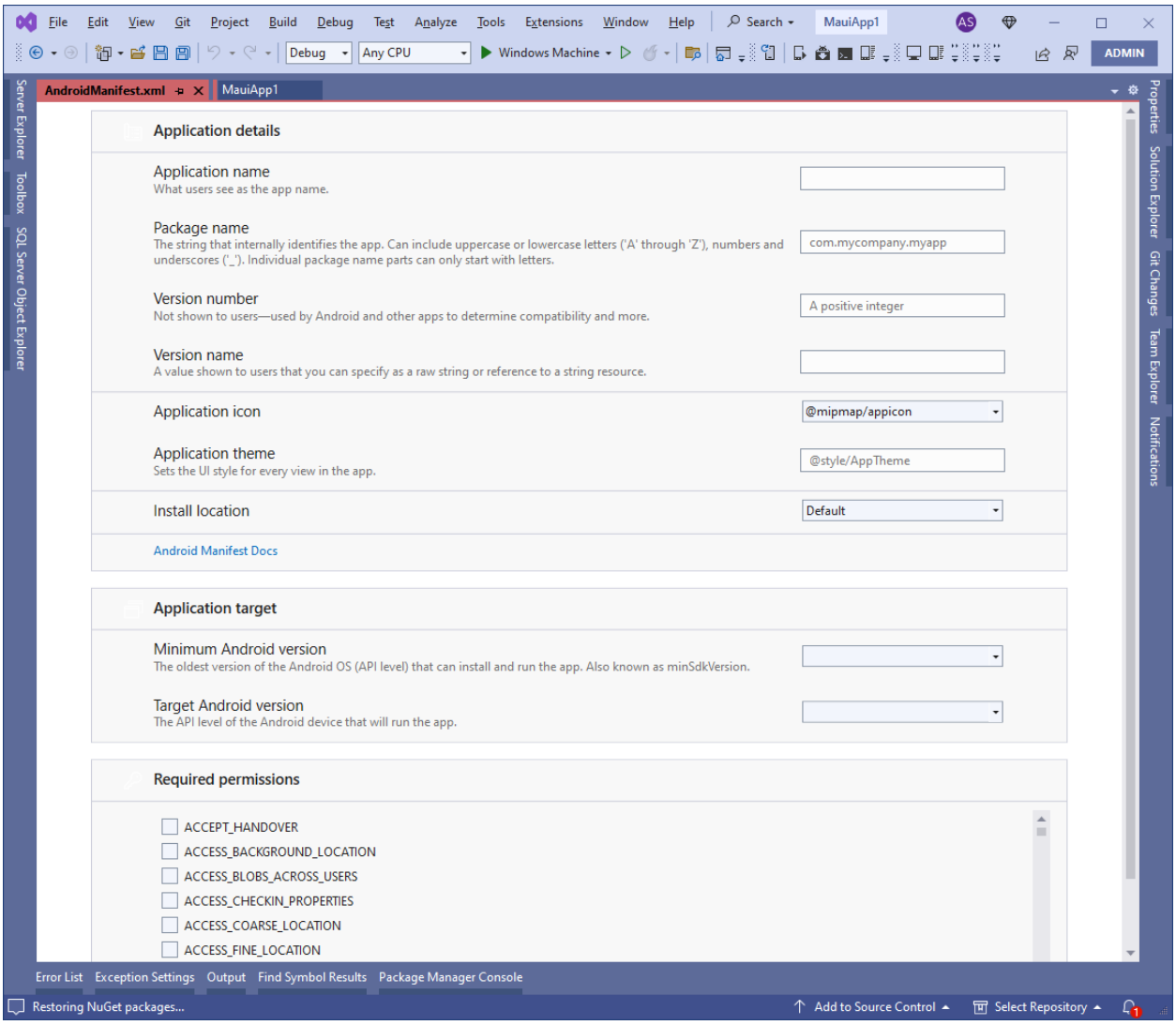

Figure 10: Setting metadata in the Android manifest

The Android manifest is very important. In it you specify your app's metadata, such as name, version number, icon, and permissions the user must grant to the application. The information you supply in the Android manifest is also important for publication to Google Play.

For example, the package name uniquely identifies your app package in the Google Play Store and, by convention, it follows the same naming described in the shared properties. The version name is your app version, whereas version number is a single-digit string that represents updates. For instance, you might have version name 1.0 and version number 1, version name 1.1 and version number 2, version name 1.2 and version number 3, and so on.

The **Install location** option allows you to specify whether your app should be installed only in the internal storage or if memory cards are allowed, but remember that starting from Android 6.0, apps can no longer be installed onto a removable storage device. In the **Minimum Android version** dropdown list, you can select the minimum Android version you want to target.

In the **Options** group, you will be able to manage debugging and build options. However, I will not walk through all the available options here. It is worth highlighting the **Use Fast Deployment** option though, which is enabled by default. When enabled, deploying the app to a physical or emulated device will only replace changed files. The **Package Signing** group allows you to sign the .apk (Android Package Kit) binary with KeyStore detail. This is also covered in the official documentation.

Configuring iOS properties

The last tab is for the iOS properties. Among all the properties, the most important is the **Bundle Signing** (see Figure 11). You use the iOS Bundle Signing properties to specify the identity that the Apple tools must use to sign the app package and to specify the provisioning profile that is used to associate a team of developers to an app identifier.

Configuring signing identities and profiles is particularly important when preparing an app for publishing.

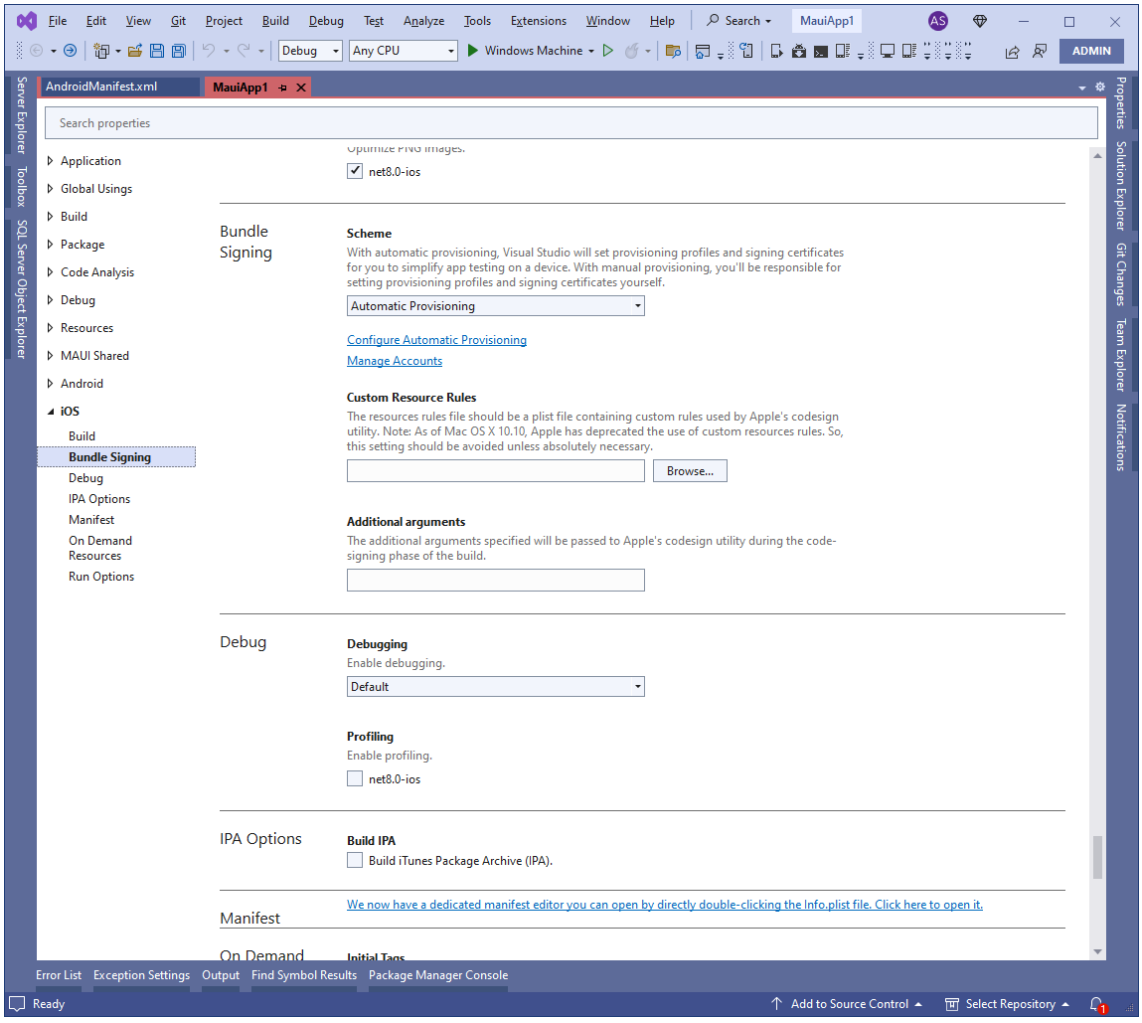

Figure 11: The iOS properties with focus on bundle signing

With the Automatic Provisioning option, you just need to enter the Apple ID associated with an active Apple Developer account and Visual Studio will generate all the provisioning profiles for you.

Click the **Manage Accounts** hyperlink and submit your Apple ID. Figure 12 shows how you supply your Apple ID and choose an identity.

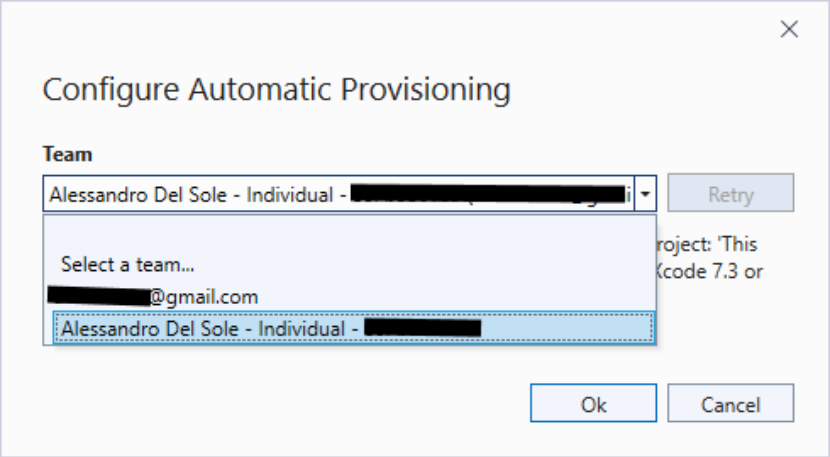

Figure 12: Configuring automatic provisioning

With the Manual Provisioning option, you will be able to adjust your settings manually. As you can imagine, Visual Studio can detect available signing identities and provisioning profiles only when connected to a Mac because this information is generated via Xcode.

The Android subfolder

The **Android** subfolder contains code and resources specific to Android devices, such as the **MainActivity.cs** file that represents the startup activity for the Android app generated by MAUI. In Android, an activity can be thought of as a single screen with a user interface, and every app has at least one.

In this file, Visual Studio adds startup code that you should not change. The **MainApplication.cs** file represents the main entry point with an instance of the `MauiProgram` class, which is discussed later in this chapter. The **AndroidManifest.xml** file represents the Android manifest described previously. The **Resources** folder is also very important because it allows you to include platform-specific resources.

Note: The same concepts apply to the Tizen subfolder.

The iOS and MacCatalyst folders

For iOS and Mac Catalyst, the project uses the **iOS** and **MacCatalyst** subfolders. The **AppDelegate.cs** file contains the MAUI initialization code and should not be changed. In the **Info.plist** (property list) file (see Figure 13), and through each tab, you can configure your app metadata, the minimum target version, supported devices and orientations, capabilities (such as Game Center and Apple Maps integration), and other advanced features.

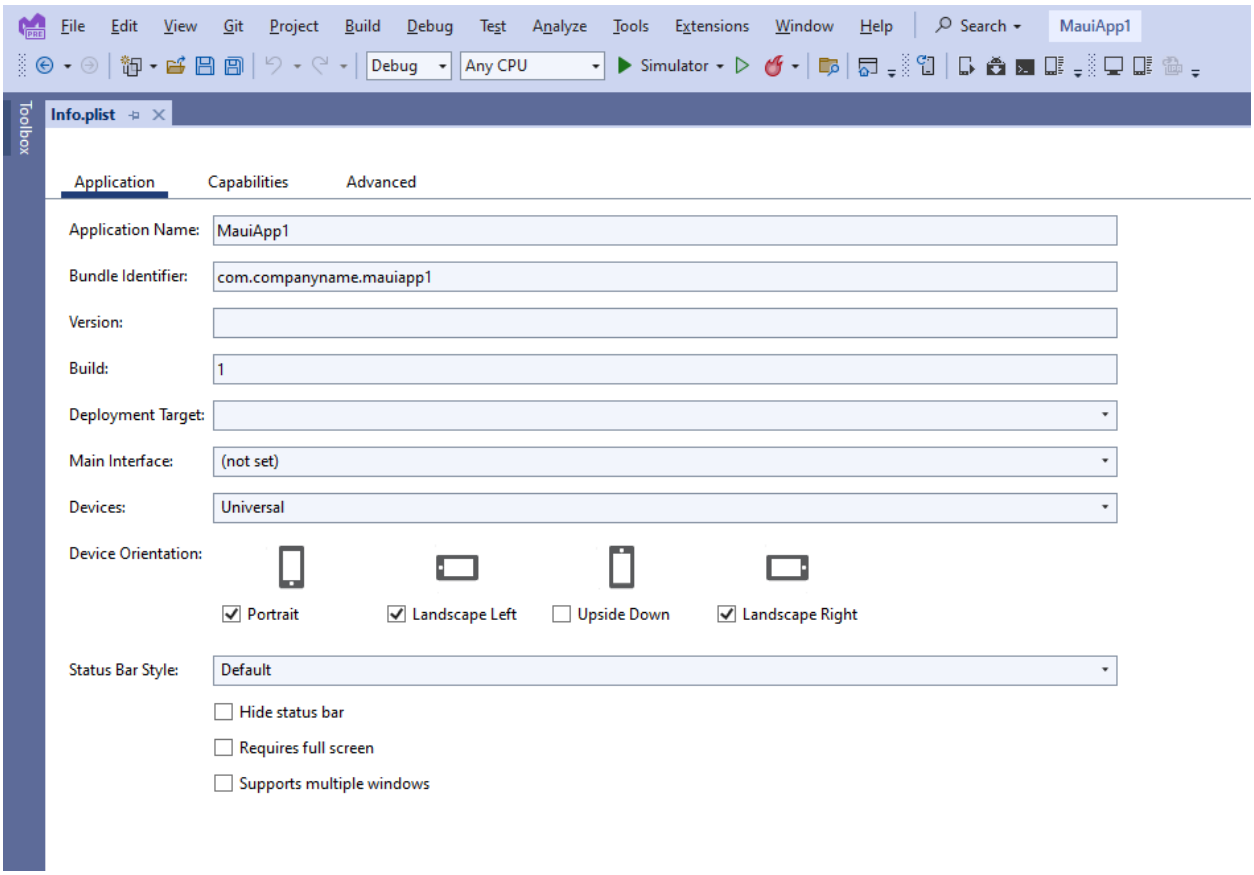

Figure 13: The Info.plist editor

The **Info.plist** file represents the app manifest in iOS, and therefore, is not related to only .NET MAUI. In fact, if you have experience with Xcode and native iOS development, you already know this file. The Info.plist reference will help you understand how to properly configure this file, especially for advanced features. You will need to set it for both iOS and Mac Catalyst if you intend to target both platforms.

The Windows folder

The Windows folder contains files that are related to the WinUI code generation to support Windows 10 and 11. In the **App.xaml.cs** file, you can see initialization code that you must not change. What you will need to configure instead is the application manifest, which you can edit by double-clicking the **Package.appxmanifest** file in Solution Explorer.

This allows you to configure app metadata (see Figure 14), visual assets such as icons and logos, and capabilities. These include permissions you need to specify before testing and distributing your apps, and Windows will ask the user for confirmation before accessing resources that require permission.

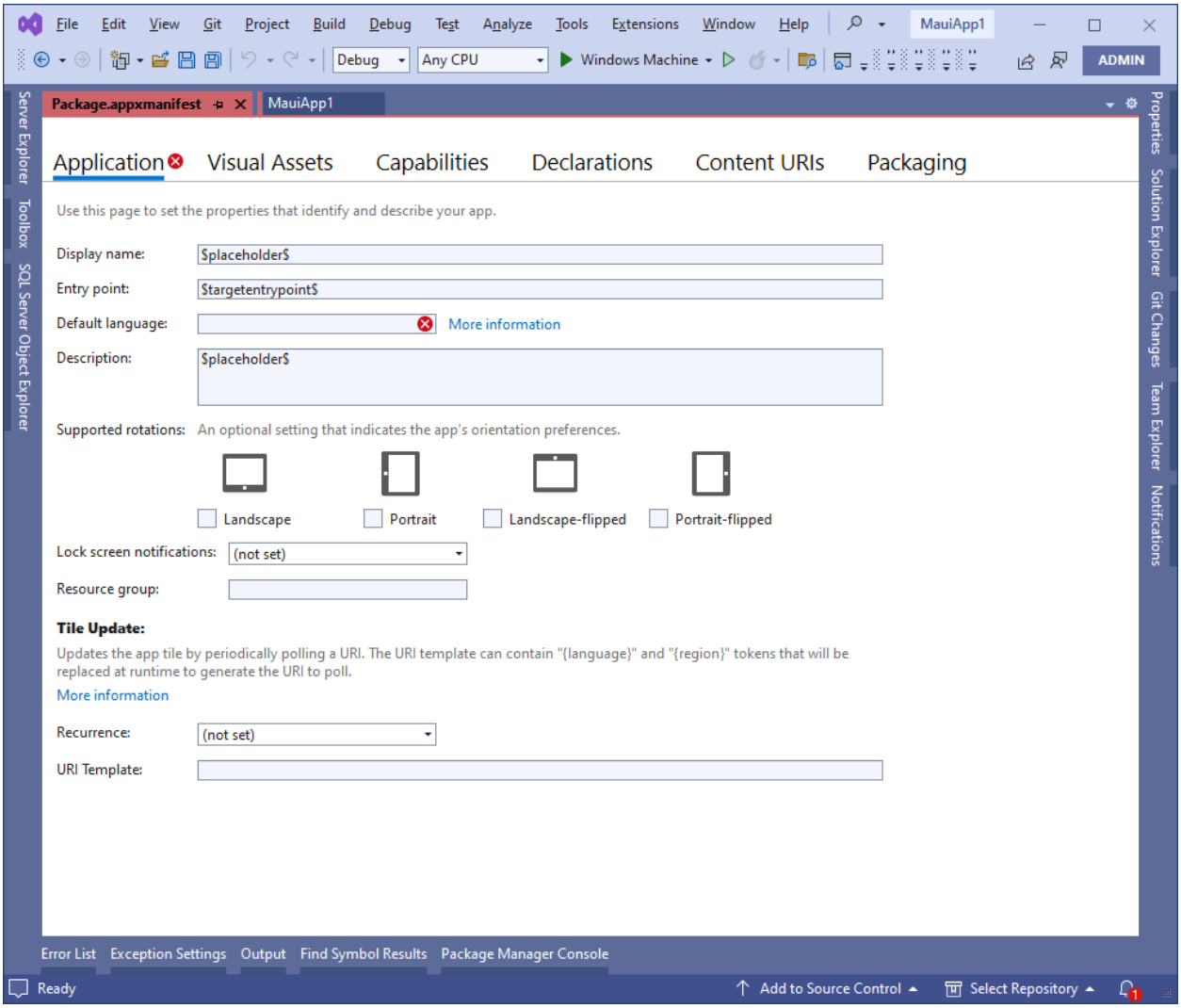

Figure 14: Editing WinUI metadata

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

The MauiProgram class

The **`MauiProgram`** class represents the entry point of a MAUI project, and its code is shown in Code Listing 1.

Code Listing 1

```
public static class MauiProgram
{
	public static MauiApp CreateMauiApp()
	{
```

```
            var builder = MauiApp.CreateBuilder();
            builder
                  .UseMauiApp<App>()
                  .ConfigureFonts(fonts =>
                  {
                        fonts.AddFont("OpenSans-Regular.ttf",
                                    "OpenSansRegular");
                        fonts.AddFont("OpenSans-Semibold.ttf",
                                    "OpenSansSemibold");
                  });
#if DEBUG
               builder.Logging.AddDebug();
#endif

            return builder.Build();
      }
}
```

The **MauiProgram** class defines one static method called **CreateMauiApp** that returns an object of type **MauiApp**. This one represents a running instance of the application. In order to return an instance of this object, the code invokes the **MauiApp.CreateBuild** method, which first returns an object of type **MauiAppBuilder**. This object exposes methods that you can invoke to configure the app. For instance, the **ConfigureFonts** method you see in Code Listing 1 is used to register fonts.

> ***Tip: Fonts, like handlers and image sources, can be configured in the MauiProgram class once, without the need to do it for every platform like you had to do in Xamarin.Forms.***

Once resources have been registered, the **Build** method of the **MauiAppBuilder** class is invoked to return an instance of the **MauiApp** class. The **CreateMauiApp** method is then invoked by each platform's startup code. Notice how a debug logger is used when the debug configuration is detected. This is useful to send logging messages to the Output window of Visual Studio.

Debugging and testing applications locally

Starting apps built with .NET MAUI for debugging and testing is easy. You simply select one of the supported platforms from the standard toolbar, and then you select the target device and press **F5**, as shown in Figure 15.

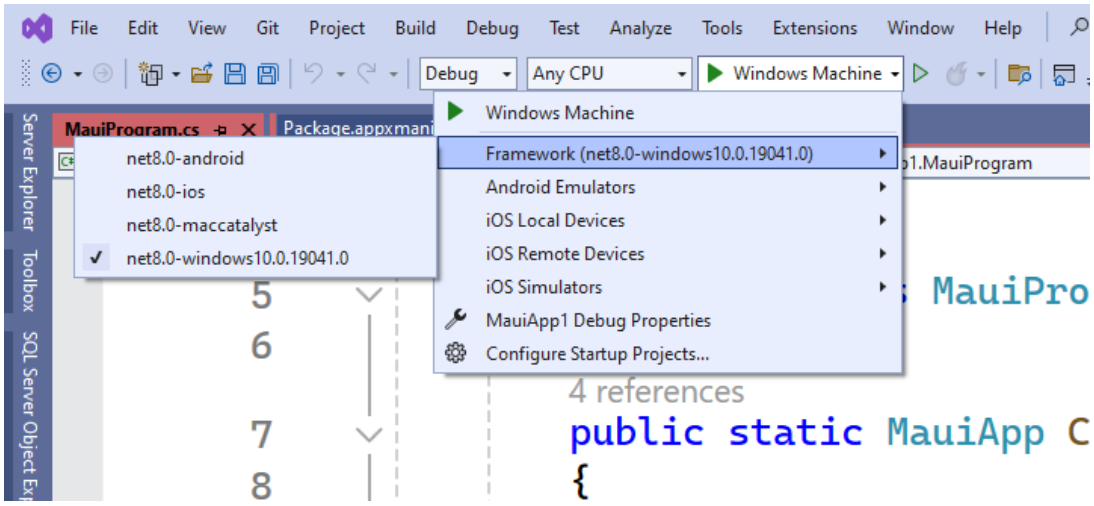

Figure 15: Selecting target platform and devices for debugging

You have two options:

- Expand the **Framework** group and select the target platform you want to debug for. This will use the default device or simulator, and it is the only option you have for debugging on Windows.
- Expand the group of each platform and select one of the connected devices or installed simulators.

Do not forget to rebuild your solution before debugging for the first time. When you start an app for debugging, Visual Studio will build your solution and deploy the app package to the selected physical device or emulator.

When the app starts (either on a physical device or an emulator), Visual Studio attaches an instance of the debugger, and you will be able to use all the well-known, powerful debugging tools of the IDE, including (but not limited to) breakpoints, data tips, tool windows, watch windows, and more.

Figure 16 shows the previously created blank app running on iOS and Android platforms within the respective emulators. Notice how in the background Visual Studio shows the **Output** window where you receive messages from the debugger; you will be able to use all the other debugging tools similarly.

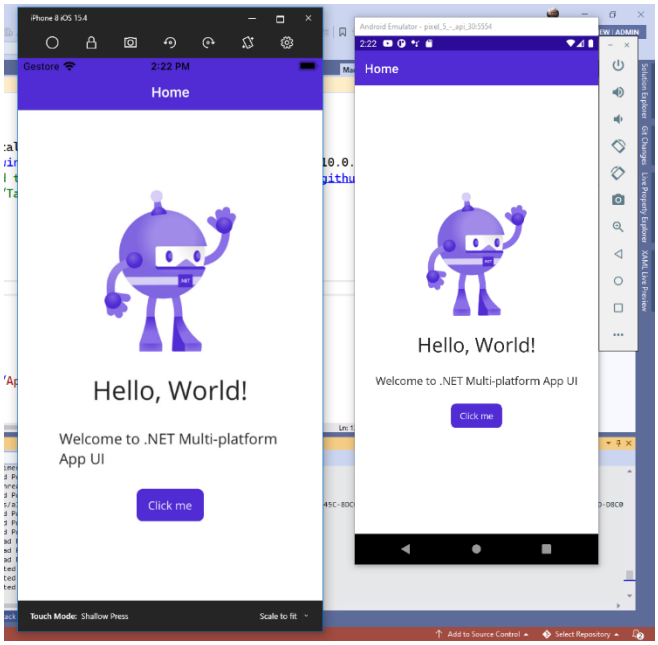

Figure 16: An app built with MAUI running on Android and iOS

As you can imagine, in order to build the iOS app, Visual Studio was connected to a Mac and launched the Apple SDKs.

Setting up the iOS Simulator

Unlike the Android and Windows emulators that run locally on your Windows development machine, the iOS Simulator runs on your Mac. However, in Visual Studio 2022, you can mirror the Simulator to Windows.

In VS 2022, open **Tools** > **Options** > **Xamarin** > **iOS Settings**, and select the **Remote Simulator to Windows** option. At the time of writing, this option is still located under the Xamarin settings.

Running apps on physical devices

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

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

At this point, you can simply plug your device into the USB port of your PC and Visual Studio will immediately recognize it. It will be visible in the list of available devices.

For Windows, you need to enable developer mode as well. To accomplish this, follow these steps:

1. Open the **Start** menu.
2. Search for **Developer Settings** and double-click it to open.
3. Turn on the **Developer Mode** option.
4. Click **Yes** to accept the warning message, which informs you that untrusted code will be able to run on your machine.

Then you will be able to plug your devices into the USB port of your PC, and Visual Studio will recognize them as available target devices.

For Apple mobile devices, you need to connect your iPhone or iPad to your Mac computer and make them discoverable through Xcode. Then, when you start debugging from Visual Studio, your app will be deployed to the iPhone or iPad through the Mac.

Tips about publishing apps

The software development lifecycle is made of different phases, such as design, development, test, and deployment. This book mostly focuses on development and cannot go into the details of publishing apps to stores due to the length and complexity of the topic.

The .NET MAUI documentation has a section called **Publish** that walks through all the necessary steps to publish apps to the different supported platforms. Just remember that you need paid accounts on the various stores to submit your apps.

Chapter summary

This chapter introduced the .NET MAUI platform and its goals, describing the required development tools and an overview of a MAUI solution, covering the platform properties and their most important settings.

You have also seen how to start an app for debugging using emulators. Now that you have an overview of .NET MAUI and the development tools, the next step is understanding what is at its core: sharing code across platforms.

Chapter 2 Sharing Code Among Platforms

.NET MAUI allows you to build apps that run on Android, iOS, Mac Catalyst, Tizen, and Windows from a single C# codebase. This is possible because, at its core, MAUI allows the sharing among platforms of all the code for the user interface, and all the code that is not platform-specific. This chapter explains how code sharing works and provides suggestions on how you can build class libraries for MAUI.

Code sharing in .NET MAUI

In the previous chapter you saw how to create a .NET MAUI solution, and you walked through the project system. The structure of a .NET MAUI project makes it very simple to write code once and run it on all the supported platforms at no additional cost.

The main reason for this, and the biggest difference with Xamarin.Forms, is that a .NET MAUI project is based on .NET 8, which is the latest version of the .NET technology.

If you expand the **Dependencies** node of a .NET MAUI project, you will see that there are four flavors of .NET 8, each targeting a specific platform, but all with the same core set of APIs. Figure 17 shows these dependencies.

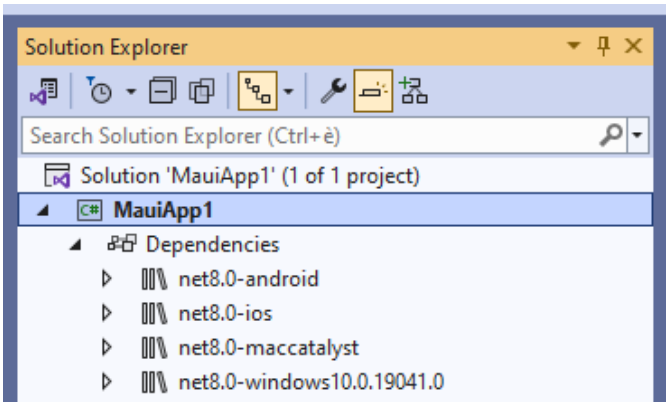

Figure 17: Dependencies of a .NET MAUI project

With a common set of APIs, your project can easily target multiple platforms. At debugging time, MAUI knows how to resolve the appropriate .NET 8 flavor depending on the target device. At deployment time, Visual Studio resolves and packages the appropriate dependencies based on the target platform.

In summary, all the code you write will run on all the supported platforms. You still have the option to add customizations by writing platform-specific code in those cases where you need access to specific system APIs or views. This will be the topic of Chapter 9.

Working with libraries

You can extend the codebase of your .NET MAUI solutions with both NuGet packages and class libraries. In both cases, you must keep in mind that:

- If the library only contains reusable code, it must be built on .NET 8.
- If the library also contains views, assets, resources, and platform-specific code, it must be built with support to .NET MAUI.

Visual Studio 2022 provides a specific project template to create reusable class libraries for .NET MAUI, called .NET MAUI Class Library, that you can see back in Figure 5. When you create a .NET MAUI library, either as a stand-alone project or as an addition to a .NET MAUI solution, Visual Studio generates a project structure that is similar to the application template. Figure 18 shows how the structure appears in Solution Explorer.

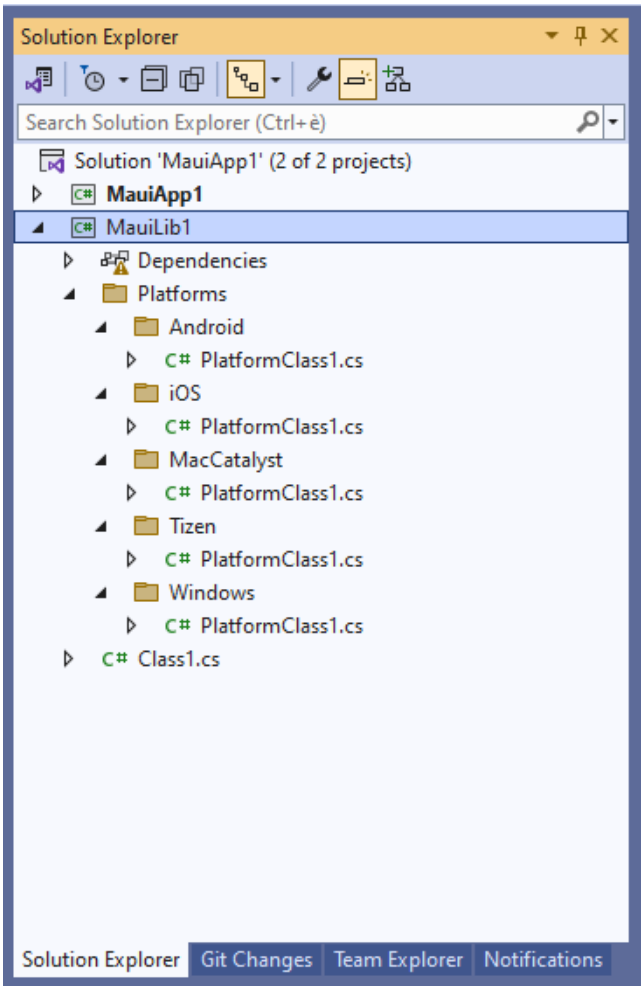

Figure 18: Structure of a .NET MAUI class library

As you can see, a .NET MAUI class library has the same framework dependencies of the .NET MAUI application project, plus an explicit reference to the .NET 8 general framework. It also provides platform subfolders where you will be able to add code that needs to run only on the selected systems. Do not forget to add a reference to the new library from the MAUI app project.

Tip: Visual Studio 2022 allows you to quickly generate NuGet packages for your libraries. In the project properties, you can enable the *Generate NuGet package on build* ***option in the*** *Package* ***tab, providing the necessary metadata. When you build the solution, a NuGet package is also built. This is not limited to .NET MAUI libraries; it is also available to .NET libraries.***

Chapter summary

This chapter discussed code sharing in .NET MAUI and explained how code sharing works because of .NET 8. Also, you had a look at reusable class libraries with .NET MAUI as the specific target. Now that you have basic knowledge of how code sharing works and why it is fundamental, in the next chapters you will start writing code and building cross-platform user interfaces.

Chapter 3 Building the User Interface with XAML

At its core, .NET MAUI is a managed layer that allows you to create native user interfaces from a single C# codebase by sharing code. This chapter provides the foundations for building the user interface in a .NET MAUI solution. Then, in the next three chapters, you will learn about layouts, controls, pages, and navigation in more detail. If you have previous experience with Xamarin.Forms, you will definitely be familiar with all the discussed concepts.

The structure of the user interface in .NET MAUI

One of the benefits of .NET MAUI is that you can define the entire user interface of your application inside a single project. Native apps for iOS, Android, Mac, Tizen, and Windows that you get when you build a solution will render the user interface with the proper native layouts and controls on each platform. This is possible because MAUI maps native controls into C# classes that are then responsible for rendering the appropriate visual element, depending on the platform the app is running on. These classes represent visual elements such as pages, layouts, and views.

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

In Chapter 9, you will learn how to extend your app capabilities by using native controls, but for now, let's focus on how .NET MAUI allows the creation of user interfaces with visual elements provided out of the box.

The user interface in iOS, Android, Tizen, Mac, and Windows has a hierarchical structure made of pages that contain layouts that contain controls. Layouts can be considered as containers of controls that allow for dynamically arranging the user interface in different ways. Based on this consideration, .NET MAUI provides numerous page types, layouts, and controls that can be rendered on each platform.

When you create a .NET MAUI solution, the project will contain a root page that you can populate with visual elements. Then you can design a more complex user interface by adding other pages and visual elements. To accomplish this, you can use both C# and Extensible Application Markup Language (XAML). Let's discuss both methods.

Coding the user interface in C#

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

Code Listing 2

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

var newLayout = new VerticalStackLayout();
newLayout.Padding = new Thickness(10);

var newLabel = new Label();
newLabel.Text = "Welcome to .NET MAUI!";

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

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

newPage.Content = newLayout;
```

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

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

You have a much more versatile way of designing the user interface with XAML, as you'll learn in the next section. Obviously, there are still situations in which you might need to create visual elements in C#; for example, if you need to add new controls at runtime (although this is the only scenario for which I suggest you code visual elements in C#).

Note: It is also possible to use C# Markup, a new set of so-called fluent APIs that allow for writing the user interface in C# with a simplified approach. C# Markup will not be covered in this book because it is only available through the .NET MAUI Community Toolkit library. However, it is covered in the final chapter of my ebook .NET MAUI Community Toolkit Succinctly.

Designing the user interface with XAML

As its name implies, XAML is a markup language that you can use to write the user interface definition in a declarative fashion. XAML was first introduced more than 15 years ago with Windows Presentation Foundation, and it has always been available in platforms such as Silverlight, Windows Phone, the Universal Windows Platform, and Xamarin.Forms.

XAML derives from XML and, among others, it offers the following benefits:

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

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

Note: XAML in .NET MAUI adheres to Microsoft's XAML 2009 specifications, but its vocabulary is different from XAML in other platforms, such as WPF or UWP. So, if you have experience with these platforms, you will notice many differences in how visual elements and their properties are named. Also, remember that XAML is case-sensitive for object names and their properties and members.

For example, when you create a .NET MAUI solution, you will find a file called MainPage.xaml, whose markup is represented in Code Listing 3.

Code Listing 3

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiApp2.MainPage">

    <ScrollView>
        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">

            <Image
                Source="dotnet_bot.png"
                SemanticProperties.Description="Cute dot net bot waving hi
to you!"
                HeightRequest="200"
                HorizontalOptions="Center" />
```

```
            <Label
                Text="Hello, World!"
                SemanticProperties.HeadingLevel="Level1"
                FontSize="32"
                HorizontalOptions="Center" />

            <Label
                Text="Welcome to .NET Multi-platform App UI"
                SemanticProperties.HeadingLevel="Level2"
                SemanticProperties.Description="Welcome to dot net Multi
platform App U I"
                FontSize="18"
                HorizontalOptions="Center" />

            <Button
                x:Name="CounterBtn"
                Text="Click me"
                SemanticProperties.Hint="Counts the number of times you
click"
                Clicked="OnCounterClicked"
                HorizontalOptions="Center" />

        </VerticalStackLayout>
    </ScrollView>
</ContentPage>
```

A XAML file in .NET MAUI normally contains a page or a custom view. The root element in Code Listing 2 is a **`ContentPage`** object, which represents its C# class counterpart and is rendered as an individual page. In XAML, the **`Content`** property of a page is implicit, meaning you do not need to write a **`ContentPage.Content`** element. The compiler assumes that the visual elements you enclose between the **`ContentPage`** tags are assigned to the **`ContentPage.Content`** property. The **`ScrollView`** allows for scrolling visual elements that require more space than the available screen size. The **`VerticalStackLayout`** is a container of views. The **`Image`** shows the specified image.

The **`Label`** object allows for displaying text. Properties of this class are assigned with XML attributes, such as **`Text`**, **`VerticalOptions`**, and **`HorizontalOptions`**. Properties can be assigned with complex types, which must be done in a hierarchical way, not inline.

You probably already have the immediate perception of better organization and visual representation of the structure of the user interface. If you look at the root element, you can see several attributes whose definitions start with **`xmlns`**. These are referred to as XML namespaces and are important because they make it possible to declare visual elements defined inside specific namespaces or XML schemas.

For example, **`xmlns`** points to the root XAML namespace defined inside a specific XML schema and allows for adding all the visual elements defined by .NET MAUI to the UI definition; **`xmlns:x`** points to an XML schema that exposes built-in types.

Each page or layout can only contain one visual element. In the case of the autogenerated MainPage.xaml page, you cannot add other visual elements to the page unless you organize them into a layout. This is the reason why, in Code Listing 3, you see several visual elements inside a **`VerticalStackLayout`**.

Let's simplify the autogenerated code to have some text and a button. This requires us to have a **`Button`** below a **`Label`**, and both must be wrapped inside a container such as the **`VerticalStackLayout`**, as demonstrated in Code Listing 4.

Tip: IntelliSense will help you add visual elements faster by showing element names and properties as you type. You can then simply press Tab or double-click to quickly insert an element.

Code Listing 4

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <VerticalStackLayout Padding="10">
        <Label Text="Welcome to .NET MAUI!"
       VerticalOptions="Center"
       HorizontalOptions="Center" />

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

</ContentPage>
```

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

If you have never seen XAML, you might wonder how you can interact with visual elements in C# at this point. In Solution Explorer, if you expand the **MainPage.xaml** file, you will see a nested file called **MainPage.xaml.cs**. This is the so-called code-behind file and it contains all the imperative code for the current page.

In this case, the simplest form of a code-behind file, the code contains the definition of the **`MainPage`** class, which inherits from **`ContentPage`**, and the page constructor, which makes an invocation to the **`InitializeComponent`** method of the base class and initializes the page.

You will access the code-behind file often from Solution Explorer, but Visual Studio offers another easy way that is related to a very common requirement: responding to events raised by the user interface.

Built-in support for accessibility

If you look at Code Listing 3, you can see that an object called **SemanticProperties** is automatically added by Visual Studio to some views. This object provides support for accessibility, especially for users with low vision.

With the **Heading** property, you can specify the hierarchical level of the view; with the **Description** property, you can specify a detailed description of the view; with the **Hint** property, you can provide a quick suggestion. If the user has enabled the system accessibility options, the device will be able to read the contents you added.

Tips about the Shell

Especially if you have experience with Xamarin.Forms, you will notice that the main page is not the startup object of a new .NET MAUI project. In fact, the startup object is the **AppShell** object, a managed representation of the Shell. The Shell is an infrastructure that offers common features to most applications, such as navigation, a flyout menu, a search bar, and more. The Shell is defined inside the AppShell.xaml file, and the basic implementation for new .NET MAUI projects looks like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    x:Class="MauiApp1.AppShell"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MauiApp2"
    Shell.FlyoutBehavior="Disabled">

    <ShellContent
        Title="Home"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />
</Shell>
```

What you need to know for now is that the Shell definition points to the **MainPage** object via another object called **ShellContent**, with a special syntax defined in the **ContentTemplate** property. Do not worry at all if this all sounds strange now, because in Chapter 6 you will learn everything about pages and the Shell.

Exactly like in Xamarin.Forms, the startup object is assigned to the **MainPage** property of the **App** class, inside the App.xaml.cs file. The following code demonstrates how this happens in a newly created .NET MAUI project:

```
public partial class App : Application
{
      public App()
      {
            InitializeComponent();
```

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

Everything will be clarified in the coming paragraphs and chapters.

Productivity features for XAML IntelliSense

The XAML code editor in Visual Studio 2022 is powered by the same engine that is behind Windows Presentation Foundation (WPF) and Universal Windows Platform (UWP). This is extremely important for several reasons:

- **Full, rich IntelliSense support**: You have linting and fuzzy matching, described in more detail shortly.
- **Quick actions and refactorings**: Light bulb suggestions are available to XAML in .NET MAUI, just as they have been for WPF, UWP, and Xamarin.Forms, making it easy to resolve XML namespaces, remove unused namespaces, and organize code with contextualized suggestions.
- **Go To Definition and Peek Definition**: These popular features from C# are also available to .NET MAUI's XAML.
- **Enhanced support for binding expressions**: IntelliSense lists available objects for bindings based on the **`{Binding}`** markup extension. It also lists available resources when using the **`{StaticResource}`** markup extension. Tips about this feature will be provided in Chapter 7.

I will now describe these productivity features in more detail.

Fuzzy matching and linting

Fuzzy matching is a feature that helps you find appropriate completions based on what you type. For example, if you type **`Stk`** and then press **Tab**, IntelliSense will add a **`StackLayout`** tag.

This feature is also capable of providing a list of possible completions as you type a control name. For example, if you type **`Layout`**, IntelliSense will offer **`VerticalStackLayout`**, **`HorizontalStackLayout`**, **`StackLayout`**, **`FlexLayout`**, **`AbsoluteLayout`**, and **`RelativeLayout`** as possible completions, as well as closing tags based on the same typing.

Another interesting feature of fuzzy matching is CamelCase matching, which provides shortcuts based on CamelCase types. For instance, if you type **`SL`** and then press **Tab**, the editor inserts a **`StackLayout`** tag. With linting, the code editor underlines code issues as you type with red squiggles (critical errors) or green squiggles (warnings).

Light bulb: quick actions and refactorings

With this tool, when a code issue is detected, you can click the light bulb icon (or press **Ctrl+.**) and IntelliSense will show potential fixes for that code issue. In .NET MAUI's XAML, the light bulb can suggest code fixes to import missing XML namespaces, sort XML namespaces, and remove unused XML namespaces.

Go To Definition and Peek Definition

Go To Definition and Peek Definition are popular and extremely useful features in the code editor, and they are also available in the XAML IntelliSense in .NET MAUI. Both are available through the context menu when you right-click an object name in the code editor.

With Go To Definition, Visual Studio will open the definition of the selected object, and if it is defined in a different file, the file will be opened in a separate tab and the cursor will be moved to the object definition. In XAML, this is particularly useful when you need to go to the definition of objects such as styles, templates, and other resources that might be defined in a different file.

Peek Definition, instead, opens an interactive pop-up window in the active editor, allowing you to see the definition of an object or make edits without leaving the active window. Additionally, you are not limited to viewing or editing objects defined in a XAML file; you can also peek the definition of an object defined in the C# code-behind file.

Figure 19 shows an example of Peek Definition where a C# event handler for the **`Clicked`** event of a **`Button`** is displayed within a XAML editor window.

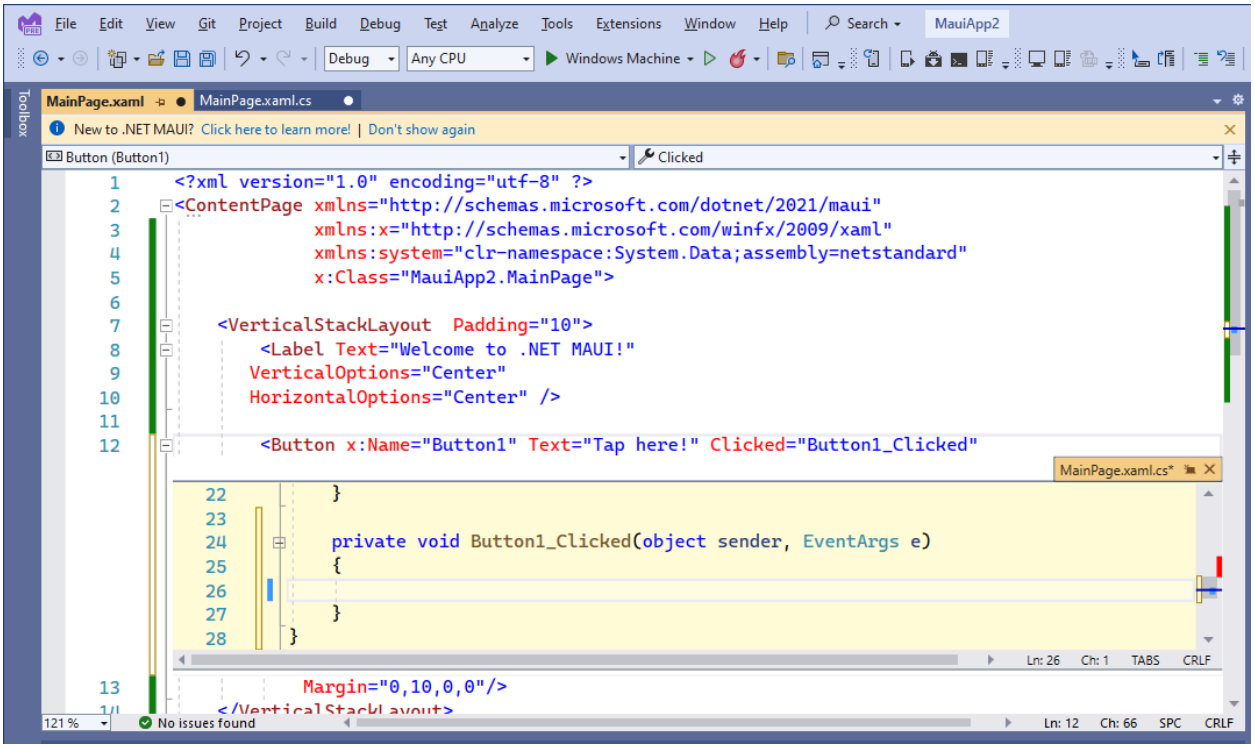

Figure 19: Making edits in the active editor with Peek Definition

Responding to events

Events are fundamental to interactions between the user and the application. Controls in .NET MAUI raise events as they normally do in any platform. Events are handled in the C# code-behind file. Visual Studio 2022 makes it simple to create event handlers with an evolved IntelliSense experience. For instance, suppose you want to perform an action when the user taps the button defined in the previous code.

The **`Button`** control exposes an event called **`Clicked`** that you assign the name of an event handler as follows:

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

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

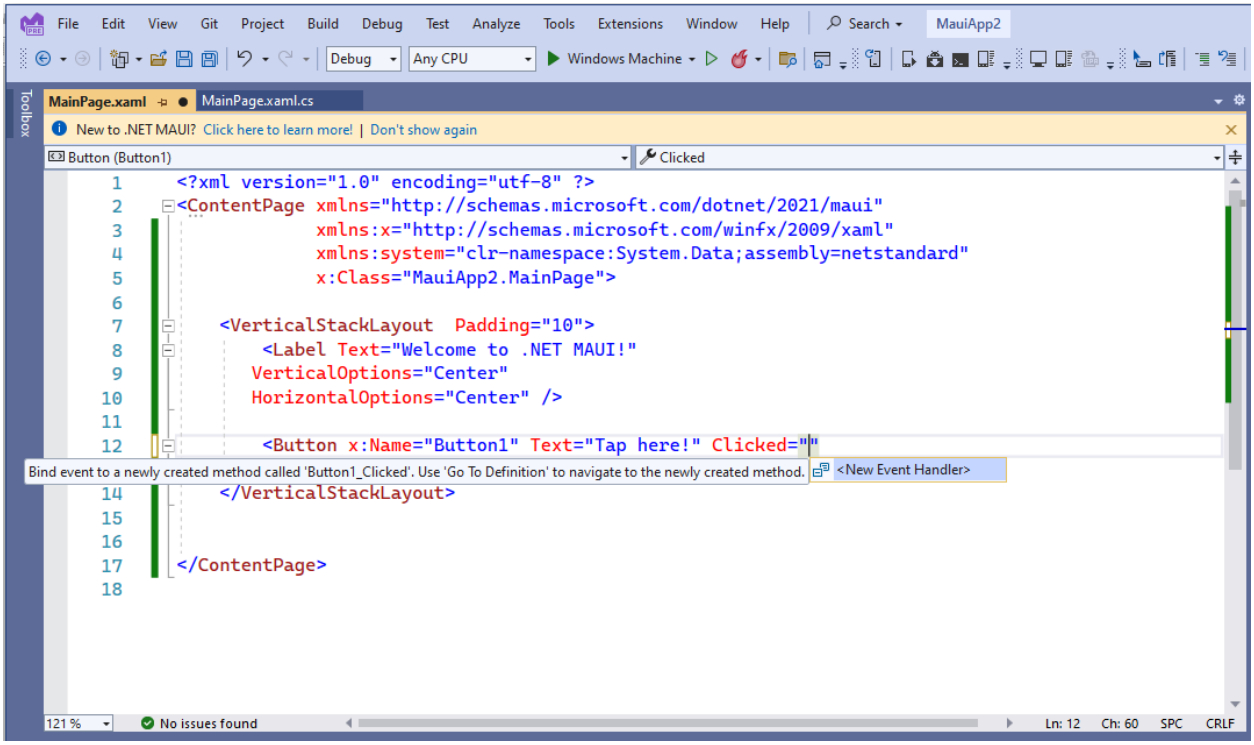

Figure 20: Generating an Event Handler

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

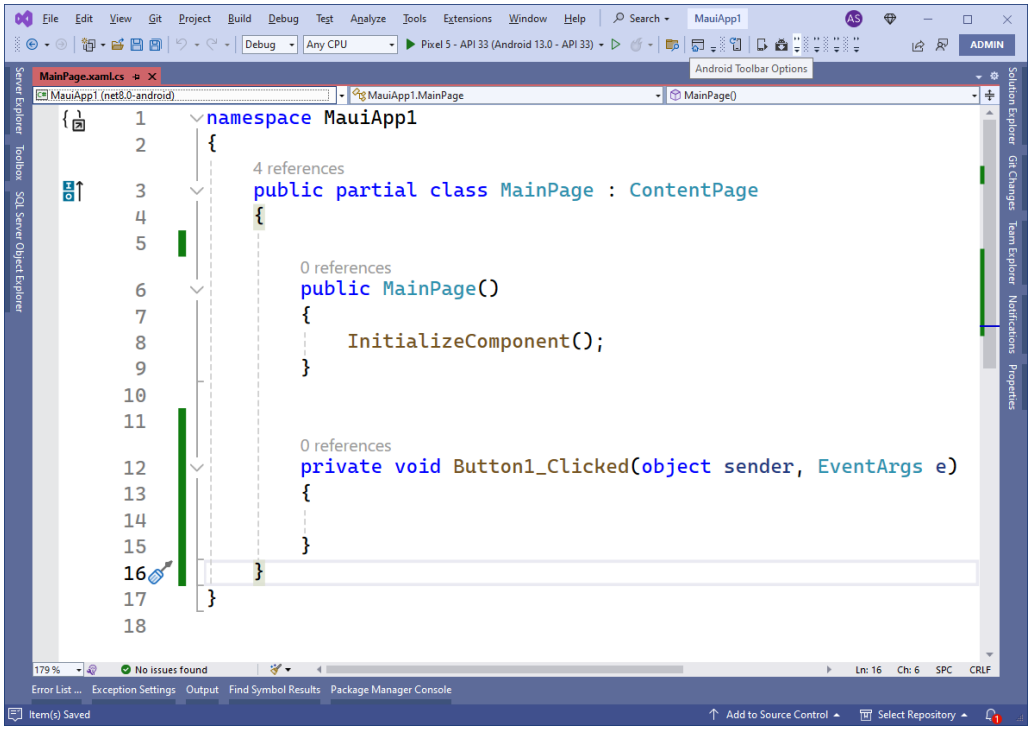

Figure 21: The event handler definition in C#

At this point, you will be able to write the code that performs the action you want to execute, exactly as it happens with other .NET platforms such as Xamarin.Forms, WPF, or UWP. Event handlers' signatures require two parameters: one of type **`object`** representing the control that raised the event, and one object of type **`EventArgs`**, containing information about the event.

In many cases, event handlers work with derived versions of **`EventArgs`**, but these will be highlighted when appropriate. As you can imagine, .NET MAUI exposes events that are commonly available on all the supported platforms.

Understanding type converters

If you look at Code Listing 2, you will see that the **`Padding`** property of the **`VerticalStackLayout`** is of type **`Thickness`**, and the **`Margin`** property assigned to the **`Button`** is also of type **`Thickness`**. However, as you can see in Code Listing 4, the same properties are assigned with values passed in the form of strings in XAML.

.NET MAUI (and all the other XAML-based platforms) implement the so-called type converters, which automatically convert a string into the appropriate value for many known types. Summarizing all the available type converters and known target types is neither possible nor necessary at this point; you simply need to remember that, in most cases, strings you assign as property values are automatically converted into the appropriate type on your behalf.

XAML Hot Reload

In the past, if you needed to make one or more changes to the XAML code, you had to stop debugging, make your changes, and then restart the application. This was one of the biggest limitations, especially with the Xamarin.Forms development experience.

In Visual Studio 2022, this problem has been solved by a feature called Hot Reload. The way it works is very simple: with the application running in debugging mode, you can make and save changes to your XAML, and the application will redraw the user interface based on your changes.

Hot Reload is available to all the supported development platforms and must be enabled by accessing **Tools** > **Options** > **Debugging** > **XAML Hot Reload**. Here you will find options to enable the feature for different platforms. Make sure that at least the **Android and iOS (.NET MAUI)** option is selected (see Figure 22).

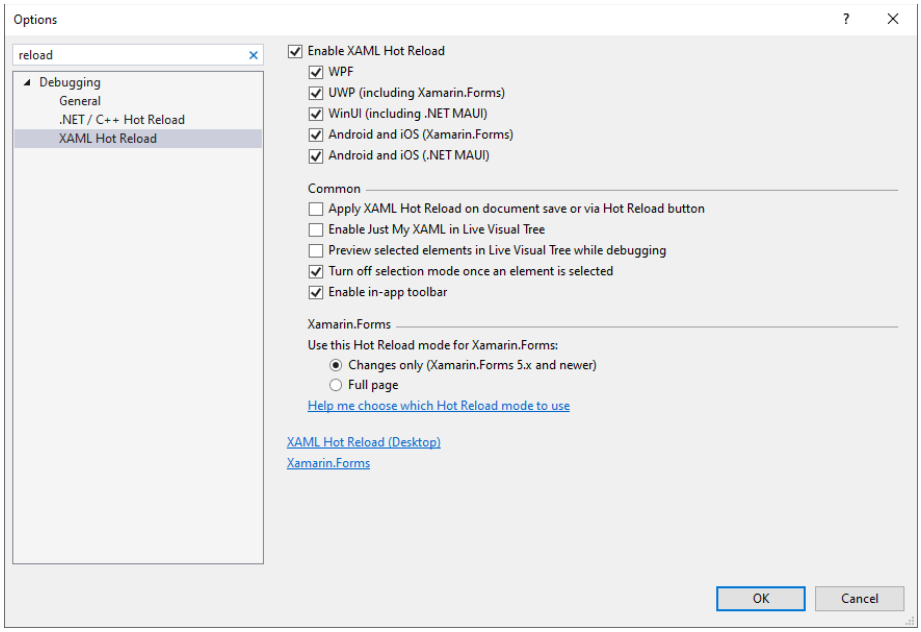

Figure 22: Enabling XAML Hot Reload

In the same dialog, you will find additional settings that are not covered here.

Chapter summary

Sharing the user interface across platforms is the main goal of .NET MAUI, and this chapter provided a high-level overview of how you define the user interface with XAML, based on a hierarchy of visual elements. You have seen how to add visual elements and how to assign their properties; you have seen how type converters allow for passing string values in XAML, and how the compiler converts them into the appropriate types.

Following this overview, it is time to discuss important UI concepts in more detail. We will start by organizing the user interface with layouts.

Chapter 4 Organizing the UI with Layouts

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

Understanding the concept of layout

> ***Tip: If you have previous experience with Xamarin.Forms, you will find a lot of similarities with layout in MAUI. If you do not have experience with Xamarin.Forms but you have worked with WPF or UWP, the concept of layout is the same as the concept of panels, such as the `Grid` and the `StackPanel`.***

One of the goals of .NET MAUI is to provide the ability to create dynamic interfaces that can be rearranged according to the user's preferences or to the device and screen size. Because of this, controls in mobile apps you build with MAUI should not have a fixed size or position in the UI, except in a very limited number of scenarios. To make this possible, controls are arranged within special containers, known as *layouts*. Layouts are classes that allow for arranging visual elements in the UI, and MAUI provides many of them.

In this chapter, you'll learn about available layouts and how to use them to arrange controls. The most important thing to keep in mind is that controls in MAUI have a hierarchical logic; therefore, you can nest multiple panels to create complex user experiences. Table 1 summarizes the available layouts. You'll learn about them in more detail in the sections that follow.

> ***Note: The documentation also includes `StackLayout` in the list of supported layouts, but at the same time, it recommends that you not use it in favor of the `VerticalStackLayout` and `HorizontalStackLayout` views. The `StackLayout` has been kept for making the migration from Xamarin.Forms easier, but it should not be used, and this is why it is not discussed here.***

Table 1: Layouts in .NET MAUI

Layout	Description
`VerticalStackLayout`	Allows you to place visual elements near each other vertically.
`HorizontalStackLayout`	Allows you to place visual elements near each other horizontally.

Layout	Description
`FlexLayout`	Allows you to place visual elements near each other horizontally or vertically. Wraps visual elements to the next row or column if not enough space is available.
`Grid`	Allows you to organize visual elements within rows and columns.
`AbsoluteLayout`	A layout placed at a specified, fixed position.
`ScrollView`	Allows you to scroll the visual elements it contains.
`Frame`	Draws a border and adds space around the visual element it contains.
`TwoPaneView`	Provides support for foldable devices.

Remember that only one root layout is assigned to the **`Content`** property of a page, and that layout can then contain nested visual elements and layouts. The **`Content`** property can also be implicit in XAML, which means that the **`<Content> </Content>`** tags can be omitted.

Note: Technically speaking, the `Frame` is a control rather than a layout, but you do not use it individually. Instead, you use it as a container of other visual elements.

Alignment and spacing options

As a general rule, both layouts and controls can be aligned by assigning the **`HorizontalOptions`** and **`VerticalOptions`** properties with one of the property values from the **`LayoutOptions`** structure, summarized in Table 2.

Providing an alignment option is very common. For instance, if you only have the root layout in a page, you will want to assign **`VerticalOptions`** with **`StartAndExpand`** so that the layout gets all the available space in the page (remember this consideration when you experiment with layouts and views in this chapter and the next one).

Table 2: Alignment Options in .NET MAUI

Alignment	Description
`Center`	Aligns the visual element at the center.

Alignment	Description
`Start`	Aligns the visual element at the left.
`End`	Aligns the visual element at the right.
`Fill`	Makes the visual element have no padding around itself, and it does not expand.

You can also control the space between visual elements with three properties: **`Padding`**, **`Spacing`**, and **`Margin`**, summarized in Table 3.

Table 3: Spacing Options in .NET MAUI

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

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

The VerticalStackLayout and HorizontalStackLayout

Both the **`VerticalStackLayout`** and **`HorizontalStackLayout`** containers allow for placing visual elements near each other as in a stack. The difference is in the orientation as their names imply. Code Listing 5 shows how to arrange visual elements with these containers.

Code Listing 5

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
```

```
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <VerticalStackLayout>
        <HorizontalStackLayout Margin="5">
            <Label Text="Sample controls" Margin="5"/>
            <Button Text="Test button" Margin="5"/>
        </HorizontalStackLayout>
        <HorizontalStackLayout Margin="5">
            <Label Text="Sample controls" Margin="5"/>
            <Button Text="Test button" Margin="5"/>
        </HorizontalStackLayout>
    </VerticalStackLayout>
</ContentPage>
```

Remember that controls within a **HorizontalStackLayout** and **VerticalStackLayout** are automatically resized according to the orientation. If you do not like this behavior, you need to specify **WidthRequest** and **HeightRequest** properties on each control, which represent the width and height, respectively.

Spacing is a property that you can use to adjust the amount of space between child elements; this is preferred to adjusting the space on the individual controls with the **Margin** property. Figure 23 shows the result of Code Listing 5.

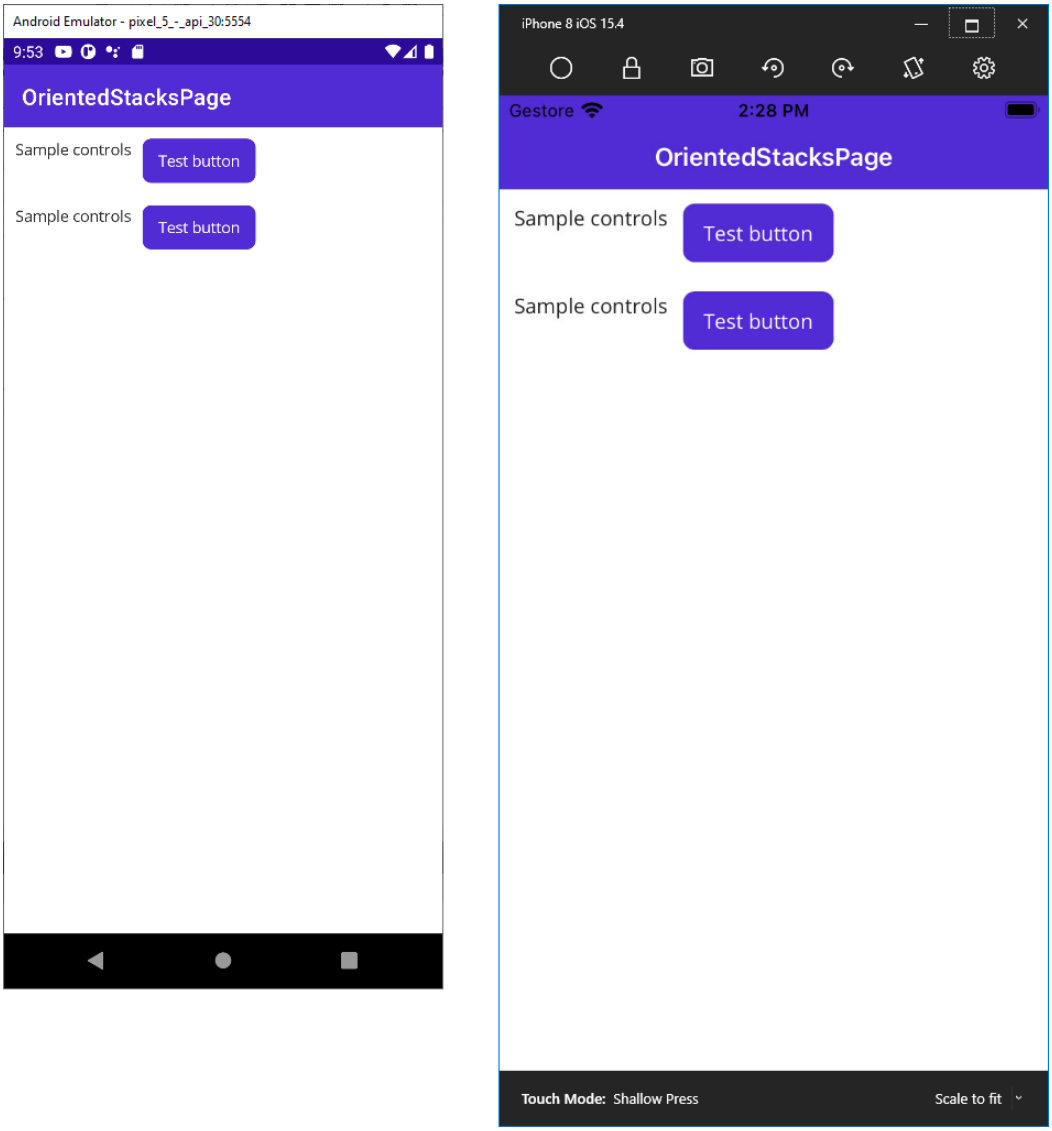

Figure 23: Arranging visual elements with the VerticalStackLayout and HorizontalStackLayout

The FlexLayout

The **`FlexLayout`** works like a **`StackLayout`** since it arranges child visual elements vertically or horizontally; the difference is that it is also able to wrap the child visual elements if there is not enough space in a single row or column. Code Listing 6 provides an example and shows how easy it is to work with this layout.

Code Listing 6

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

```
             x:Class="App1.MainPage">

        <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>
```

The **FlexLayout** exposes several properties, most of them common to other layouts, but the following are exclusive to **FlexLayout**, and certainly are the most important to use to adjust its behavior:

- **Wrap**: A value from the **FlexWrap** enumeration that specifies if the **FlexLayout** content should be wrapped to the next row if there is not enough space in the first one. Possible values are **Wrap** (wraps to the next row), **NoWrap** (keeps the view content on one row), and **Reverse** (wraps to the next row in the opposite direction).
- **Direction**: A value from the **FlexDirection** enumeration that determines if the children of the **FlexLayout** should be arranged in a single row or column. The default value is **Row**. Other possible values are **Column**, **RowReverse**, and **ColumnReverse** (where **Reverse** means that child views will be laid out in the reverse order).
- **JustifyContent**: A value from the **FlexJustify** enumeration that specifies how child views should be arranged when there is extra space around them. There are self-explanatory values such as **Start**, **Center**, and **End**, as well other options such as **SpaceAround**, where elements are spaced with one unit of space at the beginning and end, and two units of space between them, so the elements and the space fill the line; and **SpaceBetween**, where child elements are spaced with equal space between units and no space at either end of the line, again so the elements and the space fill the line. The **SpaceEvenly** value causes child elements to be spaced so the same amount of space is set between each element as there is from the edges of the parent to the beginning and end elements.

You can specify the alignment of child views in the **FlexLayout** by assigning the **FlexLayout.AlignSelf** attached property with self-explanatory values such as **Start**, **Center**, **End**, and **Stretch**. For a quick understanding, you can take a look at Figure 24, which demonstrates how child views have been wrapped.

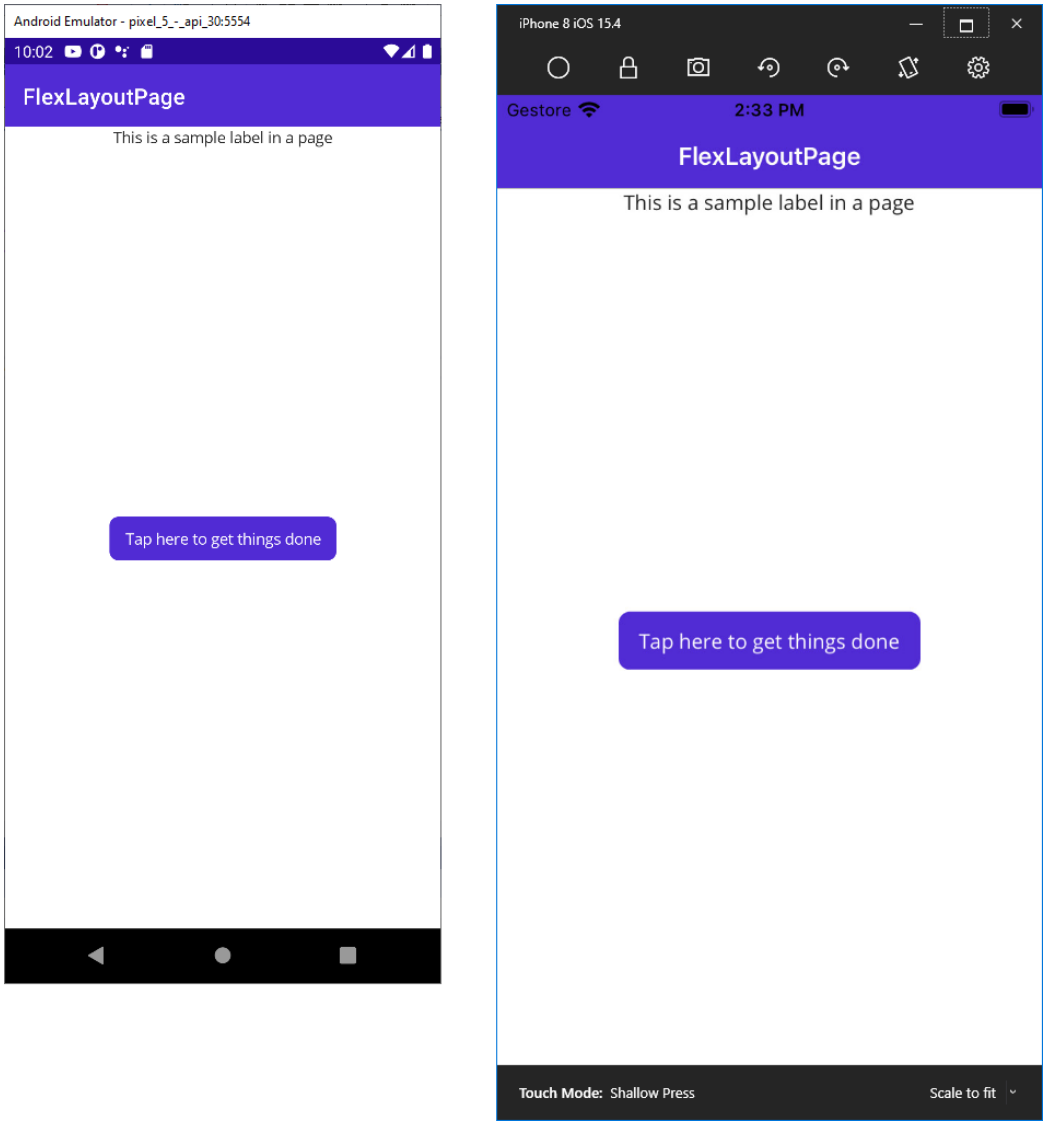

Figure 24: Arranging visual elements with the FlexLayout

If you change the **`Wrap`** property value to **`NoWrap`**, child views will be aligned on the same row, overlapping each other. The **`FlexLayout`** is therefore particularly useful to create dynamic hierarchies of visual elements, especially when you do not know the size of child elements in advance.

The Grid

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

The following code defines a **`Grid`** that is divided into two rows and two columns:

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

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

The preceding code creates a table with four cells. To place controls in the **Grid**, you specify the row and column position via the **Grid.Row** and **Grid.Column** properties, known as attached properties, on the control. Attached properties allow for assigning properties of the parent container from the current visual element.

The index of both is zero-based, meaning that **0** represents the first column from the left and the first row from the top. You can place nested layouts within a cell or a single row or column. The code in Code Listing 7 shows how to nest a grid into a root grid with child controls.

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

Code Listing 7

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             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" Margin="5"/>
            <Button Grid.Column="1" Text="Second Button" Margin="5"/>

            <Grid Grid.Row="1" Margin="10">
```

```
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Button Text="Button 3" Margin="5"/>
                <Button Text="Button 4" Margin="5" Grid.Column="1" />
            </Grid>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

Figure 25 shows the result of this code.

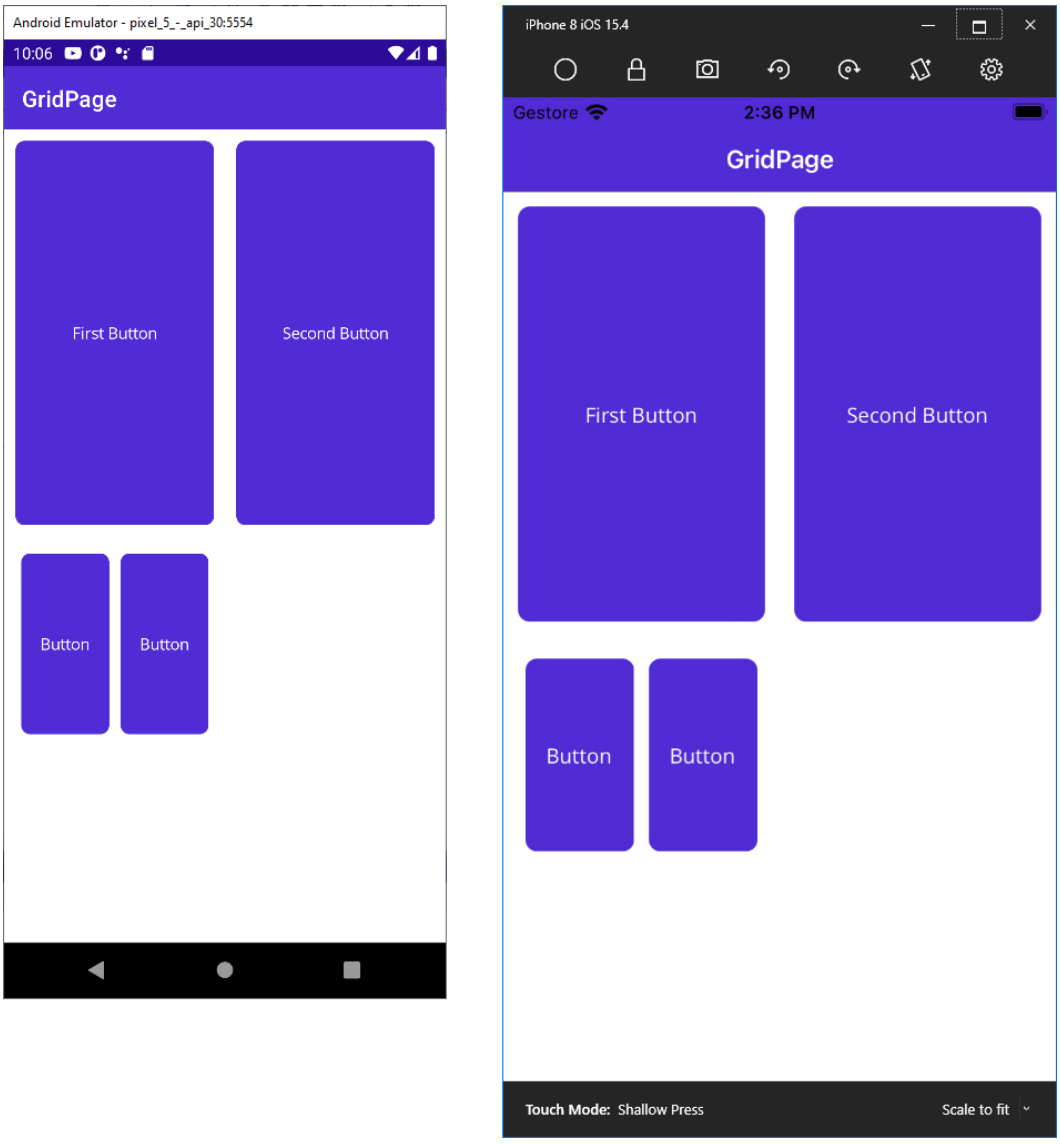

Figure 25: Arranging visual elements with the Grid

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

Spacing and proportions for rows and columns

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

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

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

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

Introducing spans

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

The AbsoluteLayout

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

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

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

Once you have created your child elements, to set them at an absolute position within the container you will need to assign the **AbsoluteLayout.LayoutFlags** attached property. You will also want to assign the **AbsoluteLayout.LayoutBounds** attached property to give the elements their bounds.

The positional values are independent of the device pixels, and the **LayoutFlags** mentioned previously support such independence. Code Listing 8 provides an example based on proportional dimensions and absolute position for child controls.

Code Listing 8

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <AbsoluteLayout Margin="10">
        <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="DarkBlue"/>
        <Label Text="Third Label"
               AbsoluteLayout.LayoutBounds="0.40, 0.40, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="DarkViolet"/>
        <Label Text="Fourth Label"
               AbsoluteLayout.LayoutBounds="0.60, 0.60, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="DarkGreen"/>
    </AbsoluteLayout>
</ContentPage>
```

Figure 26 shows the result of the **`AbsoluteLayout`** example.

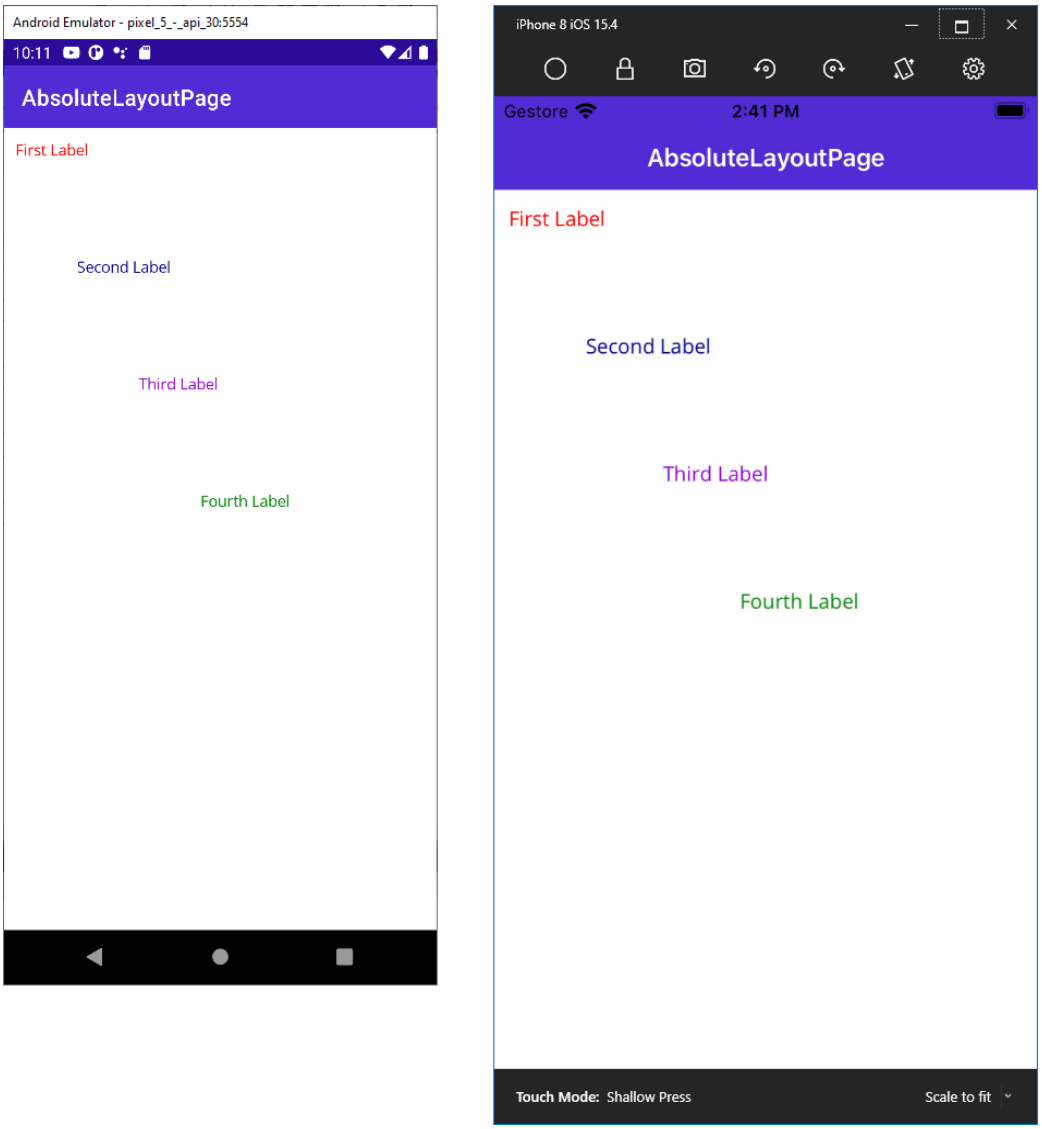

Figure 26: Absolute positioning with AbsoluteLayout

The ScrollView

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

```
<ScrollView x:Name="Scroll1">
    <StackLayout>
        <Label Text="My favorite color:" x:Name="Label1"/>
        <BoxView BackgroundColor="Green" HeightRequest="1000" />
    </StackLayout>
</ScrollView>
```

You basically add a layout or visual elements inside the **ScrollView**, and at runtime the content will be scrollable if its area is bigger than the screen size. You can also decide whether to display the scroll bars through the **HorizontalScrollbarVisibility** and **VerticalScrollbarVisibility** properties that can be assigned with self-explanatory values such as **Always**, **Never**, and **Default**.

Additionally, you can specify the **Orientation** property (with values **Horizontal** or **Vertical**) to set the **ScrollView** to scroll only horizontally or only vertically. The reason the layout has a name in the sample usage is that you can interact with the **ScrollView** programmatically, invoking its **ScrollToAsync** method to move its position based on two different options.

Consider the following lines:

```
Scroll1.ScrollToAsync(0, 100, true);
Scroll1.ScrollToAsync(Label1, ScrollToPosition.Start, true);
```

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

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

Note that you should never nest **ScrollView** layouts, and you should never include views that implement scrolling, such as the **CollectionView**.

The Frame

The **Frame** is a very special view in .NET MAUI because it is implemented as a control but used as a layout. It provides an option to draw a colored border around the visual element it contains, and optionally add extra space between the **Frame**'s bounds and the visual element. Code Listing 9 provides an example.

Code Listing 9

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">
    <Frame BorderColor="Red" CornerRadius="3"
           HasShadow="True" Margin="20">
        <Label Text="Label in a frame"
               HorizontalOptions="Center"
               VerticalOptions="Center"/>
    </Frame>
```

```
</ContentPage>
```

The **BorderColor** property is assigned with the color for the border, the **CornerRadius** property is assigned with a value that allows you to draw circular corners, and the **HasShadow** property allows you to display a shadow. Figure 27 provides an example.

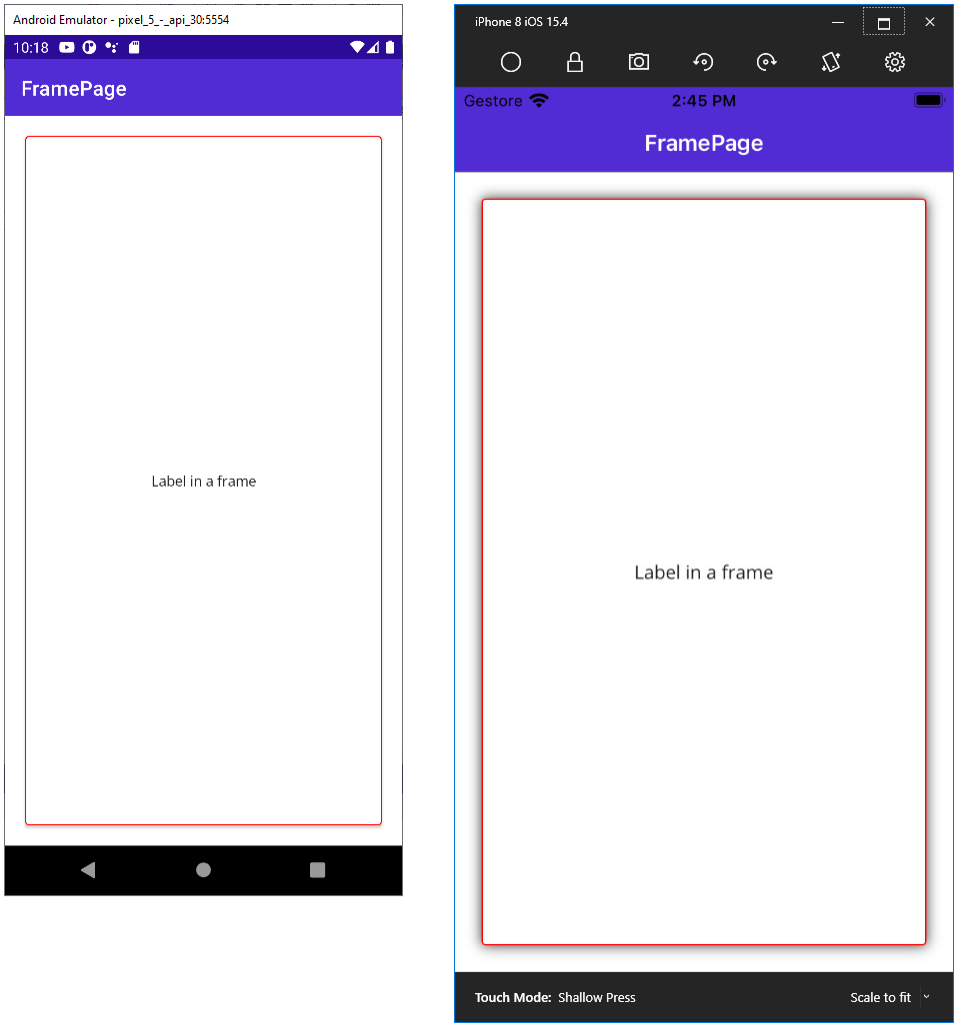

Figure 27: Drawing a Frame

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

Tip: In Xamarin.Forms, the property that represents the border color is called `OutlineColor`.

The TwoPaneView

The **TwoPaneView** is a special layout that provides support for devices with two screens (such as the Microsoft Surface Duo), generally referred to as foldable devices. It represents a container with two views that size and position content in the available space, either side-by-side or top-to-bottom. It is exposed by the same-named class, which inherits from **Grid**. This layout is not part of the .NET base class library, so you need to first install the Microsoft.Maui.Controls.Foldable NuGet package. Once you have installed this NuGet package, you need to change the **CreateMauiApp** method in the MauiProgram.cs file with the lines referenced by the comments in the following code:

```
using Microsoft.Maui.Foldable; // Add this namespace reference
...
public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();
    ...
    builder.UseFoldable(); // Add this invocation
    return builder.Build();
}
```

For Android, you then need to open the **MainActivity.cs** file located under Platforms\Android and replace the **ConfigurationChanges** assignment in the **Activity** attribute as follows:

```
ConfigurationChanges = ConfigChanges.Orientation | ConfigChanges.ScreenSize
    | ConfigChanges.ScreenLayout | ConfigChanges.SmallestScreenSize |
ConfigChanges.UiMode
```

At this point, you can use the **TwoPaneView** in the user interface of a page. You first need to add an XML namespace reference that points to the appropriate assembly, and then you need to use the layout as the root element of the page. Code Listing 10 provides an example, where you can also see how the **TwoPaneView** exposes two panes, called **Pane1** and **Pane2**.

Code Listing 10

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiApp1.FoldablePage"
             xmlns:foldable="clr-
namespace:Microsoft.Maui.Controls.Foldable;assembly=Microsoft.Maui.Controls
.Foldable"
             Title="FoldablePage">
    <foldable:TwoPaneView x:Name="twoPaneView">
        <foldable:TwoPaneView.Pane1
        BackgroundColor="#dddddd">
            <Label
```

```
            Text="Pane 1 on the first screen"
            FontSize="32"
            HorizontalOptions="Center" />
        </foldable:TwoPaneView.Pane1>
        <foldable:TwoPaneView.Pane2>
            <VerticalStackLayout BackgroundColor="LightBlue">
                <Label Text="Pane2 on the second screen"
                       TextColor="Red"/>
            </VerticalStackLayout>
        </foldable:TwoPaneView.Pane2>
    </foldable:TwoPaneView>
</ContentPage>
```

Both the **Pane1** and **Pane2** objects derive from **View**, so they can contain any visual element.

Controlling the TwoPaneView

The **TwoPaneView** supports the following three modes, but only one can be the active mode:

- **SinglePane**: Only one pane is currently visible.
- **Wide**: Both panes are laid out horizontally, one on the left and one on the right. When on two screens, this is the mode for portrait orientation.
- **Tall**: Both panes are laid out vertically, one on the top and one on the bottom. When on two screens, this is the mode for landscape orientation.

Certainly, the **TwoPaneView** can also work with one screen. If this is the case, it is possible to manage the following properties:

- **MinTallModeHeight**: This represents the minimum height the control must be to enter **Tall** mode.
- **MinWideModeWidth**: This represents the minimum width the control must be to enter **Wide** mode.
- **Pane1Length**: Sets the width of **Pane1** in **Wide** mode, the height of **Pane1** in **Tall** mode, and has no effect in **SinglePane** mode.
- **Pane2Length**: Sets the width of **Pane2** in **Wide** mode, the height of **Pane2** in **Tall** mode, and has no effect in **SinglePane** mode.

It is worth mentioning that the **PanePriority** property can be used to prefer **Pane1** or **Pane2** when in **SinglePane** mode. At the moment, there is no foldable device emulator for .NET MAUI and the hardware is not very common yet; for this reason, a sample screenshot (see Figure 28) is taken from the official documentation.

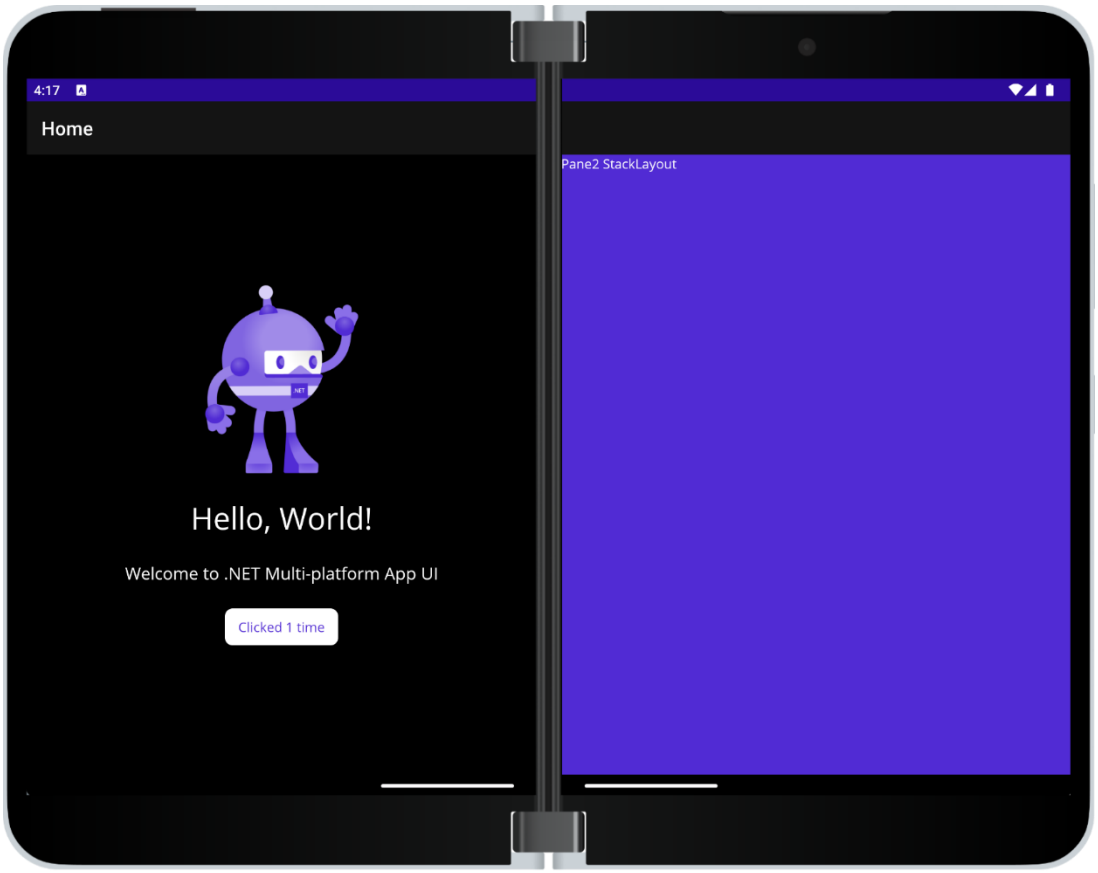

Figure 28: An example of TwoPaneView (Source: Microsoft)

Note: Emulators for foldable devices are available in the Microsoft Edge DevTools for web development, but they are not useful for nonweb development scenarios.

The ContentView

The special container **`ContentView`** allows for aggregating multiple views into a single view and is useful for creating reusable, custom controls. Because the **`ContentView`** represents a stand-alone visual element, Visual Studio makes it easier to create an instance of this container with a specific item template.

In Solution Explorer, you can right-click the project name and then select **Add New Item**. In the **Add New Item** dialog box, select the **.NET MAUI** node and then the **ContentView** item, as shown in Figure 29. Make sure you select the XAML version of the template; otherwise, you will need to supply the user interface in C# code.

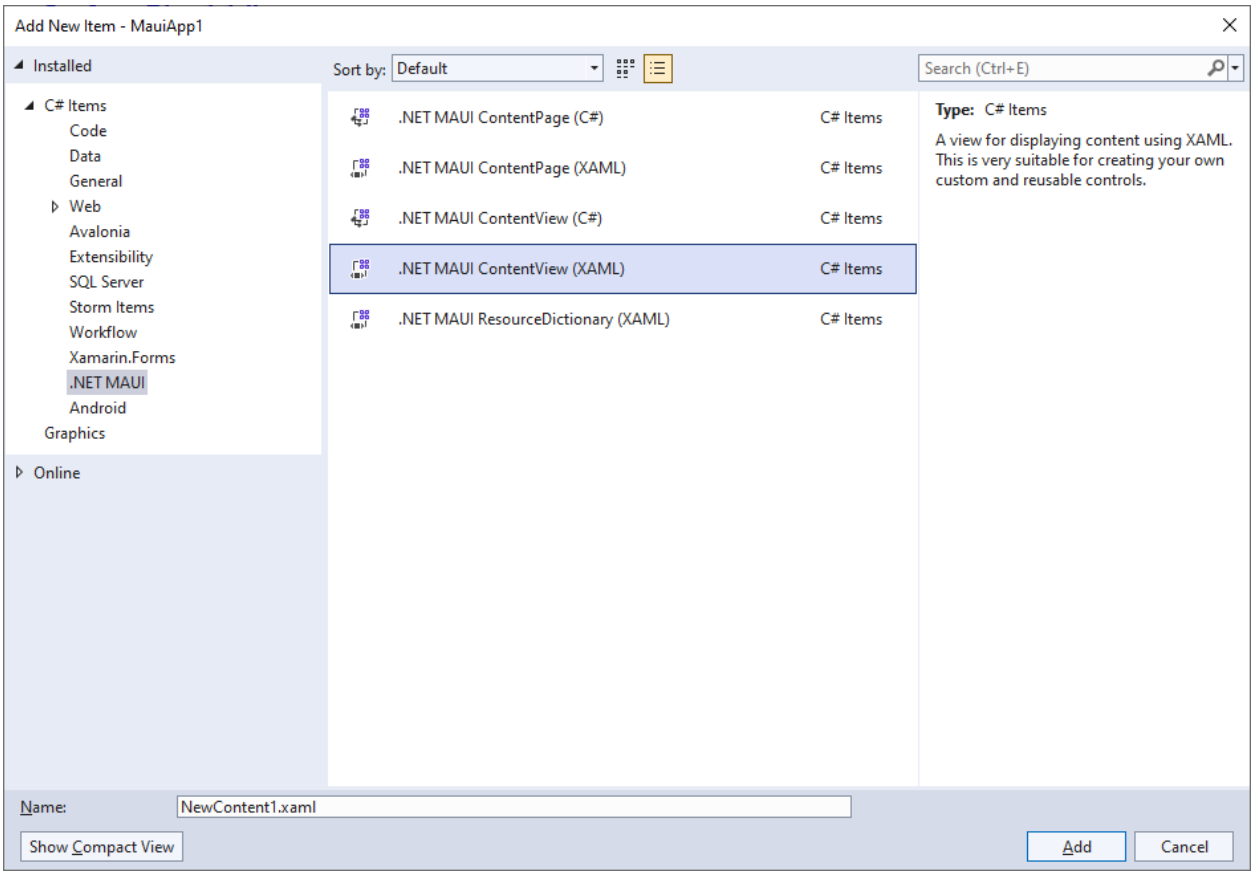

Figure 29: Adding a ContentView

When the new file is added to the project, the XAML editor shows basic content made of the **`ContentView`** root element and a **`Label`**. You can add multiple visual elements, as shown in Code Listing 11, and then you can use the **`ContentView`** as you would with an individual control or layout.

Code Listing 11

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentView xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.View1">
  <ContentView.Content>
      <VerticalStackLayout>
          <Label Text="Enter your email address:" />
          <Entry x:Name="EmailEntry" />
      </VerticalStackLayout>
  </ContentView.Content>
</ContentView>
```

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

Styling the user interface with CSS

Like its predecessor, .NET MAUI allows you to style the user interface with Cascading Style Sheets (CSS). If you have experience with creating content with HTML, you might find this feature very interesting.

> ***Note: CSS styles must be compliant with .NET MAUI in order to be consumed in mobile apps, since it does not support all CSS elements. For this reason, the feature should be considered as a complement to XAML, not a replacement. Before you decide to do serious styling with CSS in your mobile apps, make sure you read the documentation for further information about what is available and supported.***

There are three options to consume CSS styles in a MAUI project: two in XAML, and one in C# code.

Defining CSS styles as a XAML resource

> ***Note: Examples in this section are based on the `ContentPage` object since you have not read about other pages yet, but the concepts apply to all pages deriving from `Page`. Chapter 6 will describe in detail all the available pages in .NET MAUI.***

The first way you can use CSS styles in MAUI is by defining a **`StyleSheet`** object within the resources of a page, like in the following code snippet:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Layouts.CSSsample">
    <ContentPage.Resources>
        <ResourceDictionary>
            <StyleSheet>
                <![CDATA[
^contentpage {
background-color: lightgray; }
stacklayout {
margin: 20; }
]]>
            </StyleSheet>
        </ResourceDictionary>
    </ContentPage.Resources>
</ContentPage>
```

In this scenario, the CSS content is enclosed within a **CDATA** section.

Note: Names of visual elements inside a CSS style must be lowercase.

For each visual element, you supply property values 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; }**. Notice how the root element, **contentpage** in this case, must be preceded by the **^** symbol. You do not need to do anything else because the style will be applied to all the visual elements specified in the CSS.

Consuming CSS files in XAML

The second option for consuming a CSS style in XAML is from an existing .css file. First, you need to add your .css file to the project and set its **BuildAction** property as **EmbeddedResource**. The next step is to add a **StyleSheet** object to a **ContentPage**'s resources and assign its **Source** property with the .css file name, as follows:

```
<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**.

Consuming CSS styles in C# code

The last option you have for consuming CSS styles is using 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, but 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 added to the page's resources:

```
using (var reader =
       new StringReader
           ("^contentpage { background-color: lightgray; }
             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, loading the content of a CSS style from an existing file, an example is provided by the following code snippet:

```
var styleSheet = StyleSheet.FromAssemblyResource(IntrospectionExtensions.
                 GetTypeInfo(typeof(Page1)).Assembly,
                 "Project1.Assets.mystyle.css");
this.Resources.Add(styleSheet);
```

The second snippet is a bit more complex since the file is loaded via reflection (and in fact it requires a **`using System.Reflection`** directive to import the **`IntrospectionExtensions`** object). Notice how you provide the file name, including the project name (**`Project1`**) and the subfolder (if any) name that contains the .css file.

Chapter summary

Mobile apps require dynamic user interfaces that can automatically adapt to the screen size of different device form factors. In .NET MAUI, creating dynamic user interfaces is possible through a collection of so-called layouts.

The **`HorizontalStackLayout`**, **`VerticalStackLayout`**, and **`StackLayout`** allow you to arrange controls near one another both horizontally and vertically. The **`FlexLayout`** does the same, but it is also capable of wrapping visual elements.

The **`Grid`** allows you to arrange controls within rows and columns; the **`AbsoluteLayout`** allows you to give controls an absolute position; the **`RelativeLayout`** allows you to arrange controls based on the size and position of other controls or containers; the **`ScrollView`** layout allows you to scroll the content of visual elements that do not fit in a single page; the **`Frame`** layout allows you to draw a border around a visual element; and the **`ContentView`** allows you to create reusable views.

In the last part of the chapter, you saw how you can style visual elements using CSS stylesheets, in both XAML and imperative code, but these must be compliant to .NET MAUI requirements and should only be considered as a complement to XAML (and not a replacement).

Now that you have a basic knowledge of layouts, it’s time to discuss common controls that allow you to build the functionalities of the user interface, arranged within the layouts you learned in this chapter.

Chapter 5 .NET MAUI Common Controls

.NET MAUI ships with a rich set of common controls that you can use to build cross-platform user interfaces easily, and without the need for the complexity of platform-specific features. As you can imagine, the benefit of these common controls is that they run on Android, iOS, Mac Catalyst, Tizen, and Windows from the same codebase.

In this chapter, you'll learn about common controls, their properties, and their events. Other controls will be introduced in Chapter 7, especially controls whose purpose is displaying lists of data.

Using the companion code

In order to follow the examples in this chapter, either create a brand-new .NET MAUI solution (the name is up to you) or open the solution in the **Chapter5_CommonControls** folder of the companion code repository.

If you choose the first option, every time a new control is discussed, just clean the content of the root **`ContentPage`** object in the XAML file and remove any C# code specific to a single control. As an alternative, you can add a new item of type .NET MAUI ContentPage to the project and make your edits here.

If you instead want to use the companion solution directly, every time a view is discussed, uncomment the related code in the MainPage.xaml file (and in its code-behind where necessary).

Understanding the concept of view

In .NET MAUI, a *view* is the building block of any mobile application. A view represents what you would call a widget in Android, a view in iOS, and a control in Windows. Views derive from the **`Microsoft.Maui.Controls.View`** class.

The MAUI documentation groups layouts, views, and pages under the name of *controls*, so from the terminology perspective of MAUI, you need to remember this:

- A control can be a layout, page, or view.
- A layout is a container for other visual elements.
- A view is a component that allows interaction with the device and system features.
- Pages represent the organizational hierarchy of the application.

Layouts are views themselves and derive from **`Layout`**, an intermediate object in the hierarchy that derives from **`View`** and includes a **`Children`** property, allowing you to add multiple visual elements to the layout itself. From now on, I will be using *view* and *control* interchangeably, but remember that documentation and tutorials often refer to views.

Views' common properties

Views share several properties that are important for you to know in advance. These are summarized in Table 4.

Table 4: Views' common properties

Property	Description
`HorizontalOptions`	Same as Table 2.
`VerticalOptions`	Same as Table 2.
`HeightRequest`	Of type **`double`**, gets or sets the height of a view.
`WidthRequest`	Of type **`double`**, gets or sets the width of a view.
`IsVisible`	Of type **`bool`**, determines whether a control is visible on the user interface.
`IsEnabled`	Of type **`bool`**, allows enabling or disabling a control, keeping it visible on the UI.
`GestureRecognizers`	A collection of **`GestureRecognizer`** objects that enable touch gestures on controls that do not directly support touch. These will be discussed later in this chapter.

Tip: Controls also expose the `Margin` property described in Table 3.

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

Introducing common views

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

User input with the Button

The **Button** control is certainly one of the most-used controls in every user interface. You already saw a couple examples of the **Button** previously, but here is a quick summary. This control exposes the properties summarized in Table 5, and you declare it like this:

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

Table 5: Button properties

Property	Description
Text	The text in the button.
TextColor	The color of the text in the button.
BorderColor	Draws a colored border around the button.
BorderWidth	The width of the border around the button.
CornerRadius	The radius of the edges around the button.
ImageSource	An optional image to be set near the text.

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

Introducing the ImageButton

.NET MAUI gives you the option to use the **ImageButton** view when you want to display an image instead of text. Its difference from the **Button** is that this one can display both text and a small icon by assigning both the **Text** and **ImageSource** properties. The **ImageButton** can only show an image.

Working with text: Label, Entry, and Editor

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

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

LineBreakMode allows you to truncate or wrap a long string, and can be assigned a value from the **LineBreakMode** enumeration. For example, **WordWrap** splits a long string into multiple lines proportionate to the available space. If not specified, **NoWrap** is the default.

HorizontalAlignment and **VerticalAlignment** specify the horizontal and vertical alignment for the text. The **Entry** control instead allows you to enter a single line of text, and you declare it as follows:

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

The **Placeholder** property lets you display specific text in the entry until the user types something. It is useful for explaining the purpose of the text box. When the user taps the **Entry**, the on-screen keyboard is displayed.

The appearance of the keyboard can be controlled via the **Keyboard** property, which allows you to display the most appropriate keys, depending on the **Entry**'s purpose. Supported values are **Chat**, **Email**, **Numeric**, **Telephone**, and **Number**. If **Keyboard** is not assigned, **Default** is assumed.

Additionally, **Entry** exposes the **MaxLength** property that allows you to set the max length of the text the user can enter. This control also exposes two events: **Completed**, which is fired when the users finalize the text by tapping the Return key, and **TextChanged**, which is fired at every keystroke.

You provide event handlers the usual way, as follows:

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

Entry also provides the **IsPassword** property to mask the **Entry**'s content, which you use when the user must enter a password. Another very useful property is **ReturnType**, whose value is of type **ReturnType** enumeration, which allows you to specify the text to be displayed in the Enter key of the keyboard. Possible values are **Done**, **Go**, **Next**, **Search**, **Send**, and **Default** (which sets the default text for each platform).

The combination of the **Label** and **Entry** controls is visible in Figure 30.

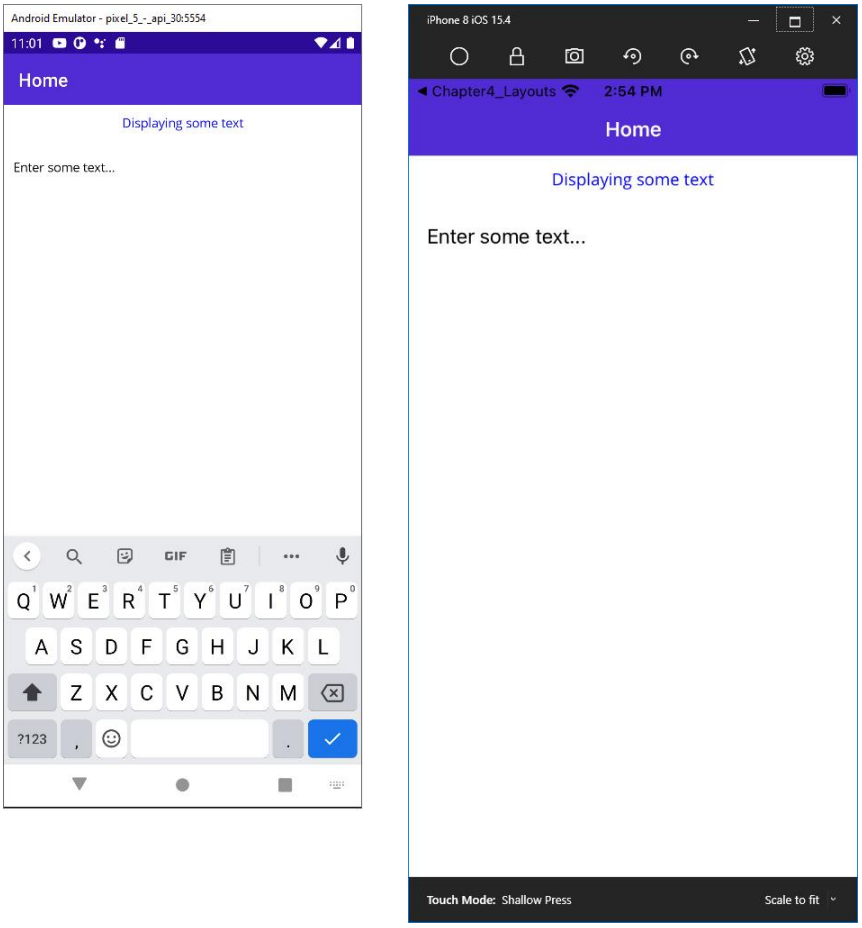

Figure 30: Label and Entry controls

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

Formatted strings and bindable spans

The **Label** control exposes the **FormattedText** property, which can be used to implement more sophisticated string formatting. This property is of type **FormattedString**, an object that is populated with a collection of **Span** objects, each representing a part of the formatted string. The following snippet provides an example:

```
<Label
   HorizontalOptions="Center"
   VerticalOptions="Center">
    <Label.FormattedText>
        <FormattedString>
            <FormattedString.Spans>
                <Span FontSize="Large" FontAttributes="Bold"
```

```
                    TextColor="Red"
                    Text=".NET MAUI Succinctly" />
                <Span FontSize="Medium" FontAttributes="Italic"
                    TextColor="Black" Text="3rd edition" />
                <Span FontSize="Small" FontAttributes="Bold"
                    TextColor="Blue"
                    Text="by Alessandro Del Sole" />
            </FormattedString.Spans>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

You can basically use **Span** objects to format specific paragraphs or sentences by setting most of the properties already exposed by the **Label** control, such as **Text**, **Font**, **FontFamily**, **FontSize**, **FontAttributes**, **TextColor**, and **BackgroundColor**.

Tip: In Xamarin.Forms, coloring text in a `Span` was accomplished via the `ForegroundColor` property.

Managing fonts

Controls that display some text (including the **Button**) or wait for user input through the keyboard also expose some properties related to fonts, such as **FontFamily**, **FontAttributes**, and **FontSize**.

FontFamily specifies the name of the font you want to use. **FontAttributes** displays text as **Italic** or **Bold** and, if not specified, **None** is assumed. With **FontSize**, you can specify the font size with either a numeric value or with a so-called named size, based on the **Body**, **Caption**, **Header**, **SubTitle**, **Title**, **Micro**, **Small**, **Medium**, and **Large** values from the **NamedSize** enumeration.

With this enumeration, .NET MAUI chooses the appropriate size for the current platform. For instance, the following two options are allowed to set the font size:

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

Unless you are writing an app for a single platform, it is recommended that you avoid using numeric values—use the named size instead. You can also manage the spacing between individual characters in a string by setting the **CharacterSpacing** property, of type **double**, whose default value is zero.

Another useful property is **TextTransform**, which you can use to automatically convert a string into uppercase or lowercase. Allowed values are **Uppercase**, **Lowercase**, **Default**, and **None**. By default, strings are not transformed. It is also possible to apply underline and strikethrough decorations to a string by assigning the **TextDecorations** property with **Underline** and **Strikethrough** (the default is **None**).

Using custom fonts

A common requirement with modern apps is to use custom fonts. These can be available through official design channels, such as Google Material, or produced by professional designers. .NET MAUI supports both .ttf and .otf file formats.

In order to use custom fonts, follow these steps:

1. Add the .ttf or .otf files into the **Resources\Fonts** subfolder of the project.
2. Set the **Build Action** property of each file to **MauiFont**.
3. Assign the **`FontFamily`** property of your view with the name of the font as follows:

   ```
   <Label Text="Some text" FontFamily="MyFont.ttf"/>
   ```

4. Register the font with the app. This is accomplished in the **MauiProgram.cs** file. When you create a new project, two fonts are also registered so you have the following reference:

   ```
   var builder = MauiApp.CreateBuilder();
   builder
         .UseMauiApp<App>()
         .ConfigureFonts(fonts =>
         {
               fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
               fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
         });
   ```

In summary, for each custom font, you invoke the **`AddFont`** method, passing the file name and the friendly name. This method works over an **`IFontCollection`** object returned by the **`ConfigureFonts`** extension method, and is assigned to the instance of the **`MauiAppBuilder`** class, an object that represents the running instance of the app.

If you worked with custom fonts in Xamarin.Forms, you know how simpler the MAUI approach is, because you only need to copy font files to one folder and assign properties just once.

Working with dates and time: DatePicker and TimePicker

Another common requirement in mobile apps is working with dates and time: .NET MAUI provides the **`DatePicker`** and **`TimePicker`** views for that. On each platform, these are rendered with the corresponding date and time selectors.

`DatePicker` exposes the **`Date`**, **`MinimumDate`**, and **`MaximumDate`** properties that represent the selected or current date, the minimum date, and the maximum date, respectively, all of type **`DateTime`**. It exposes an event called **`DateSelected`**, which is raised when the user selects a date. You can handle this to retrieve the value of the **`Date`** property. The view can be declared as follows:

```
<DatePicker x:Name="DatePicker1" MinimumDate="07/17/2021"
            MaximumDate="06/30/2022"
            DateSelected="DatePicker1_DateSelected"/>
```

Tip: Properties of type `DateTime` are immediately validated in the XAML editor. If you enter an invalid date, a red squiggle will appear under the date.

Then in the code-behind, you can retrieve the selected date like this:

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

The **`DateChangedEventArgs`** object stores the selected date in the **`NewDate`** property and the previous date in the **`OldDate`** property. Figure 31 shows the **`DatePicker`** in action.

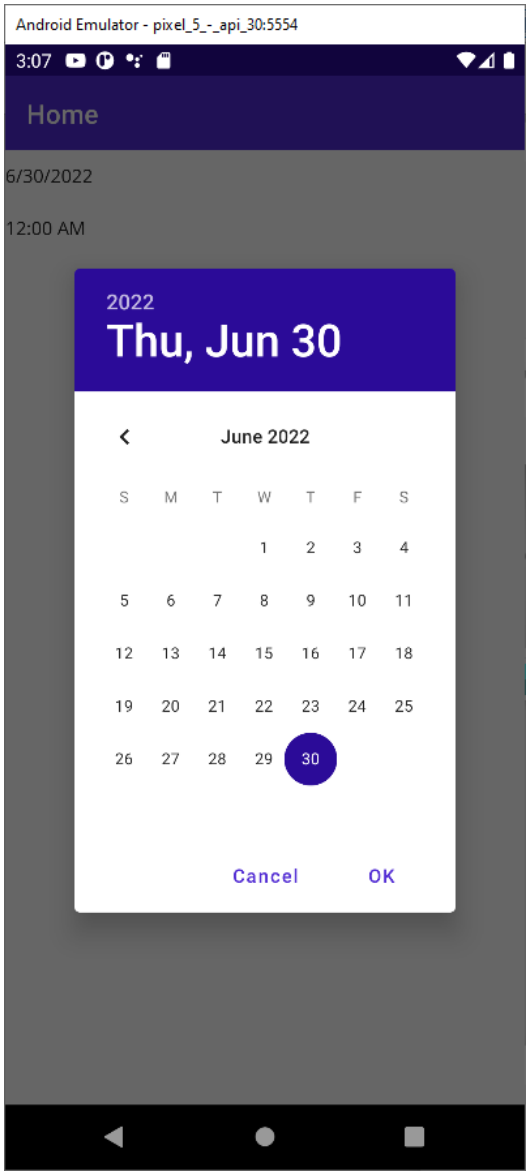

Figure 31: The DatePicker in action

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

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

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

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

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

TimeProperty is a dependency property, a concept that will be discussed in Chapter 7. Figure 32 shows the **TimePicker** in action.

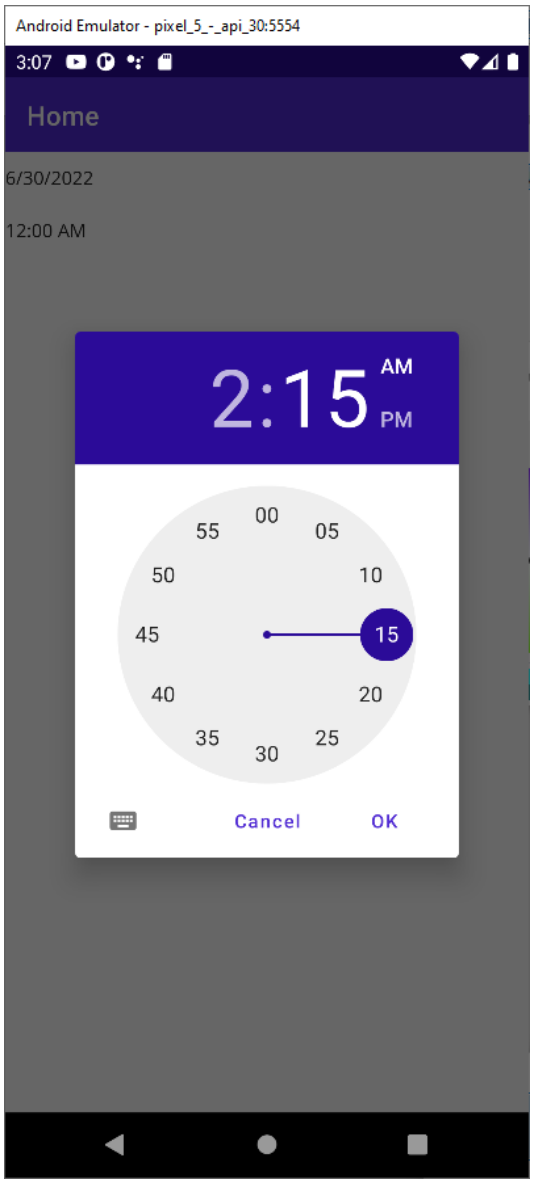

Figure 32: The TimePicker in action

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

Displaying HTML contents with WebView

The **`WebView`** control allows for displaying HTML content, including webpages and static HTML markup. This control exposes the **`Navigating`** and **`Navigated`** events that are raised when navigation starts and completes, respectively.

The real power is in its **`Source`** property, of type **`WebViewSource`**, which can be assigned with a variety of content, such as URIs or strings containing HTML markup. For example, the following XAML opens the specified website:

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

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

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

For dynamic sizing, a better option is enclosing the **`WebView`** inside a **`Grid`** layout. If you instead use one of the stack layouts, you need to supply height and width explicitly. Figure 33 shows how the **`WebView`** appears.

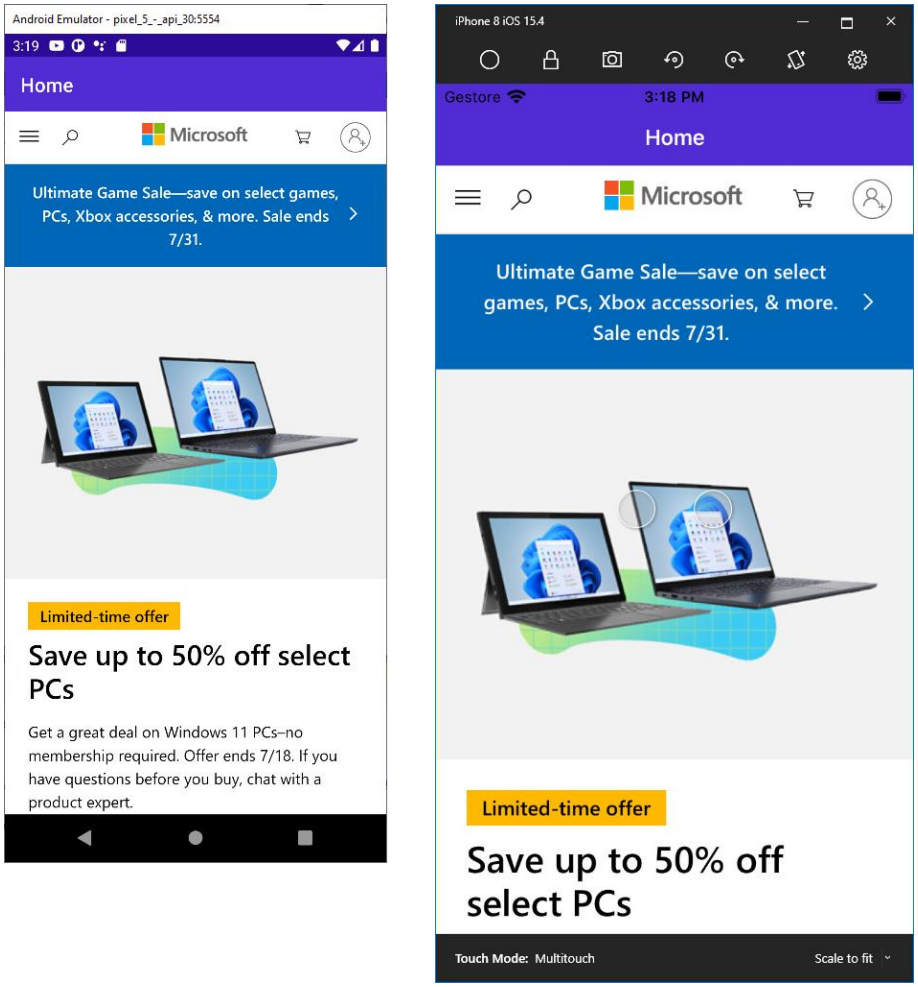

Figure 33: Displaying HTML content with WebView

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

Tips about the BlazorWebView

.NET MAUI exposes a useful control called **BlazorWebView**, which allows for hosting Blazor applications inside a MAUI project. In addition, Visual Studio 2022 includes a project template called .NET MAUI Blazor App that generates a project ready to host Blazor contents.

Using this control assumes you have built applications with Blazor, which cannot be taken for granted in this book. For this reason, here you are informed about the existence of this view, and you can refer to the Microsoft documentation for further information.

Implementing value selection

.NET MAUI offers a number of controls for user input based on selecting values. More specifically, the views provided for this purpose are the **Switch**, **CheckBox**, **RadioButton**, **Slider**, and **Stepper**, discussed in the next paragraphs.

On/off value selection with the Switch

The first of them is **Switch**, which provides a toggled value and is useful for selecting values such as true or false, on or off, and enabled or disabled. It exposes the **IsToggled** property, which turns the switch on when **true**, and the **Toggled** event, which is raised when the user changes the switch position.

This control has no built-in label, so you need to use it in conjunction with a **Label** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Enable data plan"/>
   <Switch x:Name="Switch1" IsToggled="True" Toggled="Switch1_Toggled"
           Margin="5,0,0,0"/>
</StackLayout>
```

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

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

You can also change the color for the selector when it's turned on, assigning the **OnColor** property with the color of your choice.

True/false selection with the CheckBox

The **CheckBox** is a special view that allows for selecting a true or false value via a flag. The following code demonstrates how to implement a **CheckBox**:

```
<CheckBox x:Name="CheckBox1" IsChecked="True"
          CheckedChanged="CheckBox1_CheckedChanged"/>
```

The **IsChecked** property represents the true or false selection. The **CheckedChanged** event is fired when the user changes the value of the control. The selected value is sent via an instance of the **CheckedChangedEventArgs** class as follows:

```
private void CheckBox1_CheckedChanged(object sender, CheckedChangedEventArgs e)

{

      bool selectedValue = e.Value;

}
```

The **CheckBox** is really effective on desktop systems, whereas on mobile applications a better choice in terms of user experience could be the **Switch**. Your graphic designer will help you choose the best view to fit your requirements.

Numeric value selection with the Slider

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

```
<StackLayout Orientation="Horizontal">
   <Label Text="Select your age: "/>
   <Slider x:Name="Slider1" Maximum="99" Minimum="13" Value="30"
    ValueChanged="Slider1_ValueChanged"/>
</StackLayout>
```

The **ValueChanged** event is raised when the user moves the selector on the **Slider**, and the new value is sent to the **NewValue** property of the **ValueChangedEventArgs** object you get in the event handler.

Progressive selection with the Stepper

The last control is the **Stepper**, which allows the supply of discrete values with a specified increment. It also allows the specification of minimum and maximum values. You use the **Value**, **Increment**, **Minimum**, and **Maximum** properties of type **double** as follows:

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

```
    <Label x:Name="StepperValue"/>
</StackLayout>
```

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

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

Figure 34 shows all the aforementioned views.

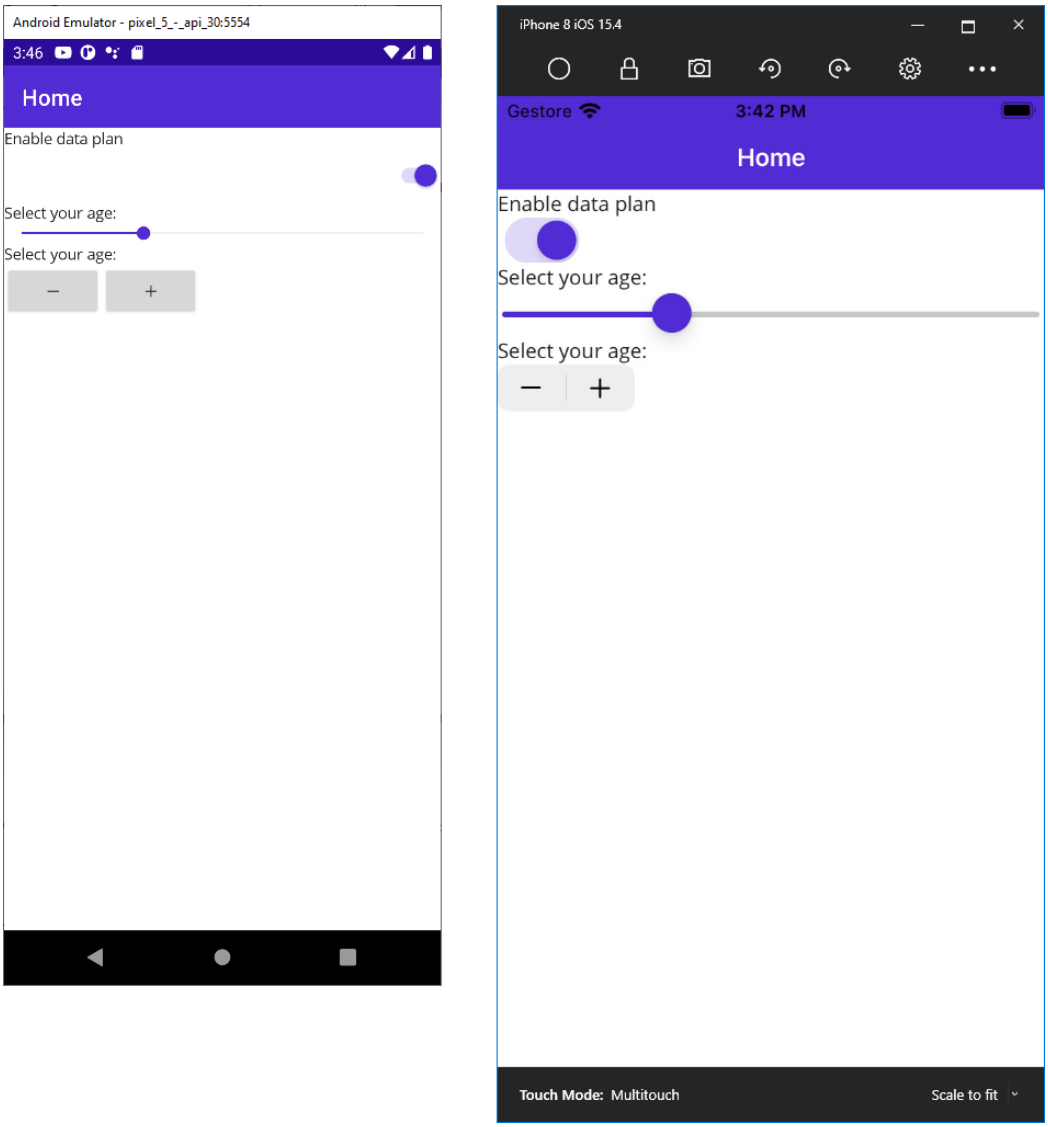

Figure 34: A summary view of the Switch, Slider, and Stepper controls

Item selection with the RadioButton

The **RadioButton** control allows for implementing mutually exclusive item selection. This means that if you have multiple options, only one will be accepted. Consider the following example:

```
<VerticalStackLayout>
    <Label Text="Where are you spending your next holidays?"/>
    <RadioButton Content="Seaside" GroupName="HolidayOptions" />
    <RadioButton Content="Countryside" GroupName="HolidayOptions"/>
    <RadioButton Content="Lake" GroupName="HolidayOptions"/>
    <RadioButton Content="Mountain" GroupName="HolidayOptions"/>
</VerticalStackLayout>
```

This allows for selecting one of the proposed values. The **Content** property is of type **IView**, which means that you are not limited to displaying strings, but you can display complex visual hierarchies (for example, images and text).

The **GroupName** property is not mandatory, but it is necessary to make radio buttons mutually exclusive to one another. All the buttons to which the property is assigned will be considered part of the same group. There is a simpler syntax based on attached properties that works as shown in the following code:

```
<VerticalStackLayout RadioButtonGroup.GroupName="HolidayOptions">
    <Label Text="Where are you spending your next holidays?"/>
    <RadioButton Content="Seaside" />
    <RadioButton Content="Countryside" />
    <RadioButton Content="Lake" />
    <RadioButton Content="Mountain" />
</VerticalStackLayout>
```

If you specify the **RadioButtonGroup.GroupName** attached property at the layout level, all the nested **RadioButton** controls will be considered part of the same group. Figure 35 shows the result of this code.

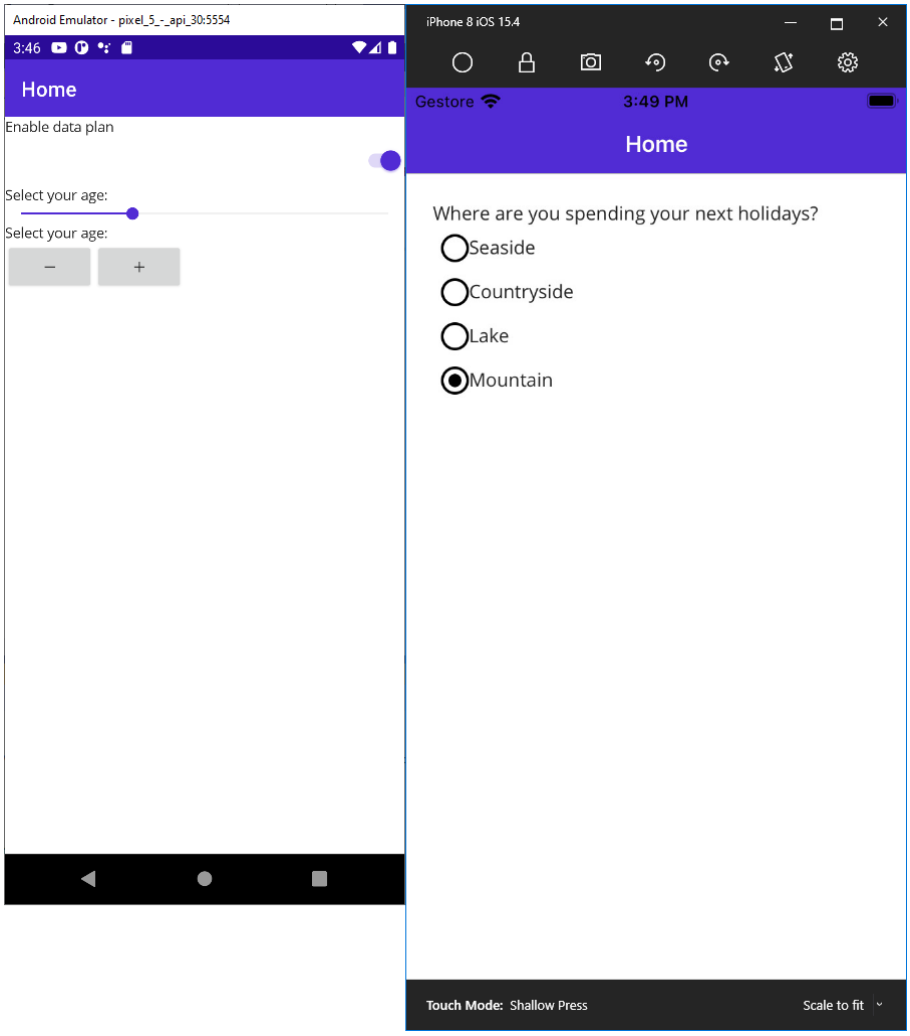

Figure 35: Item selection with RadioButton views

In the following code, you can see how to provide a complex object as the content:

```
<VerticalStackLayout Margin="20"
 RadioButtonGroup.GroupName="HolidayOptions">
    <Label Text="Where are you spending your next holidays?"/>
    <RadioButton Value="seaside">
        <RadioButton.Content>
            <HorizontalStackLayout>
                <Image Source="seaside.jpg"/>
                <Label Text="Seaside" />
            </HorizontalStackLayout>
        </RadioButton.Content>
    </RadioButton>

…

</VerticalStackLayout>
```

Especially with data-binding scenarios (such as those you will learn about in Chapter 7), it is good to assign a value to the control instead of relying on the displayed content. You can accomplish this by assigning the **`Value`** property. To handle user interactions, the **`RadioButton`** exposes the **`CheckedChanged`** event, which is fired when the status changes, and that you handle as follows:

```
void RadioButton_CheckedChanged(object sender, CheckedChangedEventArgs e)
{
    // Perform required operation
}
```

Put succinctly, the **`Value`** property, of type **`bool`**, is **`true`** when the **`RadioButton`** is selected and **`false`** when it is unselected. The same value is also stored inside the **`IsChecked`** property. The latter can be used to both read and programmatically set the value for each **`RadioButton`**.

Tip: The `RadioButton` supports control template redefinition and visual states. This is out of scope here, but you can find further information in the documentation.

Introducing the SearchBar

The **`SearchBar`** is a very useful view. It shows a native search box with a search icon that users can tap. This view exposes the **`SearchButtonPressed`** event. You can handle this event to retrieve the text the user typed in the box, and then perform your search logic; for example, by executing a LINQ query against an in-memory collection or filtering data from the table of a local database.

It also exposes the **`TextChanged`** event, which is raised at every keystroke, and the **`Placeholder`** property, which allows you to specify placeholder text like the same-named property of the **`Entry`** control. You declare it as follows:

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

Figure 36 shows an example.

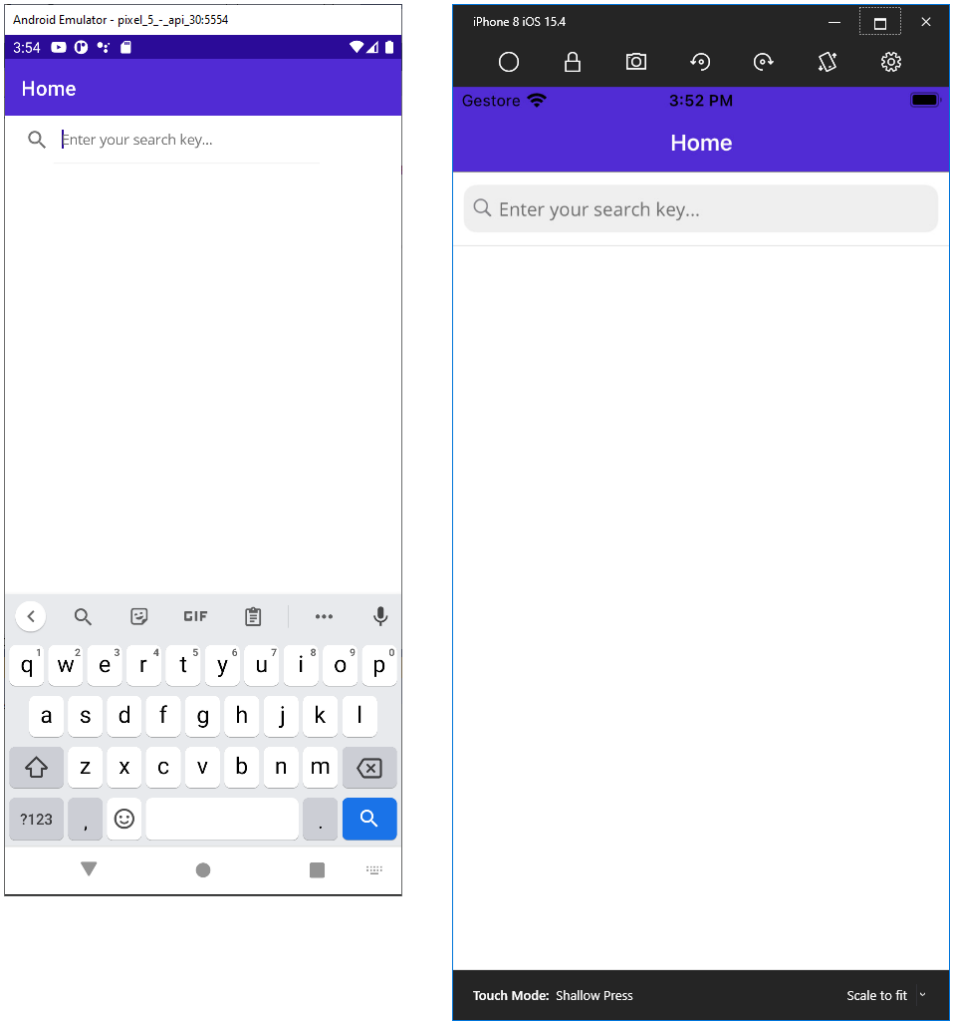

Figure 36: The SearchBar view

The **`SearchBar`** can be a good companion because you can use it to filter a list of items based on the search key the user entered, which is one of the topics discussed in Chapter 7.

Long-running operations: ActivityIndicator and ProgressBar

In some situations, your app might need to perform potentially long-running operations, such as downloading content from the internet or loading data from a local database. In such situations, it is a best practice to inform the user that an operation is in progress.

This can be accomplished with two views: the **`ActivityIndicator`** and the **`ProgressBar`**. The latter exposes a property called **`Progress`**, of type **`double`**, and a **`ProgressColor`** property, of type **`Color`**, that allows you to change the color of the progress indicator. This control is not used very often because it implies you are able to calculate the amount of time or data needed to complete an operation.

The **ActivityIndicator** shows a simple animated indicator that is displayed while an operation is running, without the need to calculate its progress. It is enabled by setting its **IsRunning** property to **true**. You might want to make it visible only when running, which is done by assigning **IsVisible** with **true**. You typically declare it in XAML as follows:

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

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

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

// Executing the operation...

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

As a personal suggestion, I recommend you always set both **IsVisible** and **IsRunning**. This will help you keep consistent behavior across platforms. Figure 37 shows an example.

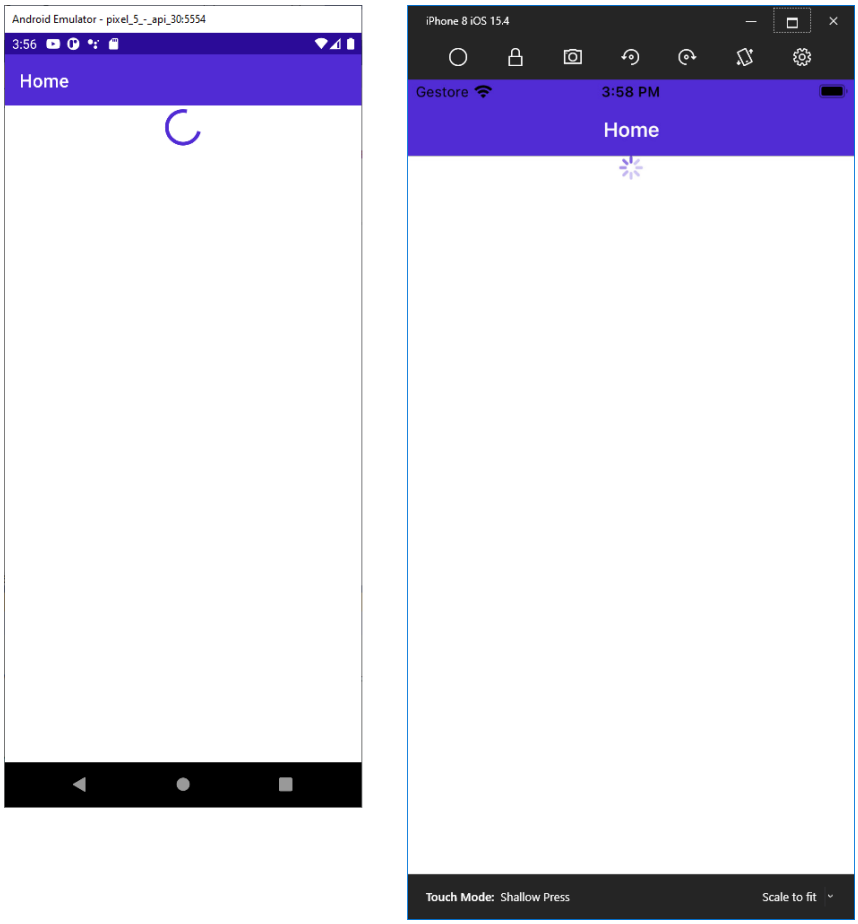

Figure 37: The ActivityIndicator shows that an operation is in progress

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

Working with images

Using images is crucial in mobile apps since they both enrich the look and feel of the user interface and enable apps to support multimedia content. .NET MAUI provides the **`Image`** control to display images from the internet, local files, and embedded resources.

Displaying images is really simple, but understanding how you load and size images is more complex, especially if you have no previous experience with XAML and dynamic user interfaces. You declare an **`Image`** as follows:

```
<Image Source="https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-cover/2018-bronze-xamarin-forms-succinctly.png" Aspect="AspectFit"/>
```

As you can see, you assign the **`Source`** property with the image path, which can be a URL or the name of a local file or resource. A convenient place to include your local images is the Resources\Images subfolder.

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

Note: The official documentation provides further explanations about images as embedded resources.

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

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

You can also set the **`WidthRequest`** and **`HeightRequest`** properties to adjust the size of the **`Image`** control. Figure 38 shows an example.

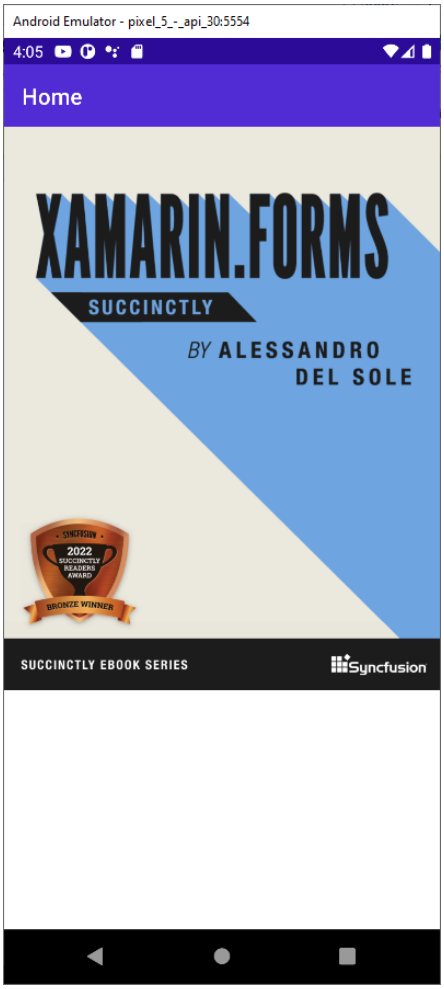

Figure 38: Displaying images with the Image view

The supported image formats are .jpg, .png, .gif, .bmp, .tif, and .svg.

Displaying animated GIFs

It is also possible to display animated GIFs. These can be displayed from local storage or streamed online. When an animated .gif file is loaded, you need to explicitly assign the **`IsAnimationPlaying`** property with **`true`**.

Displaying solid colors with the BoxView

The **`BoxView`** is a view that displays a rectangle or square filled with a solid color. It is very easy to use, as demonstrated in the following line:

```
<BoxView Color="Red" CornerRadius="20" WidthRequest="200" HeightRequest="200"
         HorizontalOptions="Center"/>
```

The size and position changes depending on the parent container. You can optionally assign the **`CornerRadius`** property if you want to add round corners. Figure 39 shows an example.

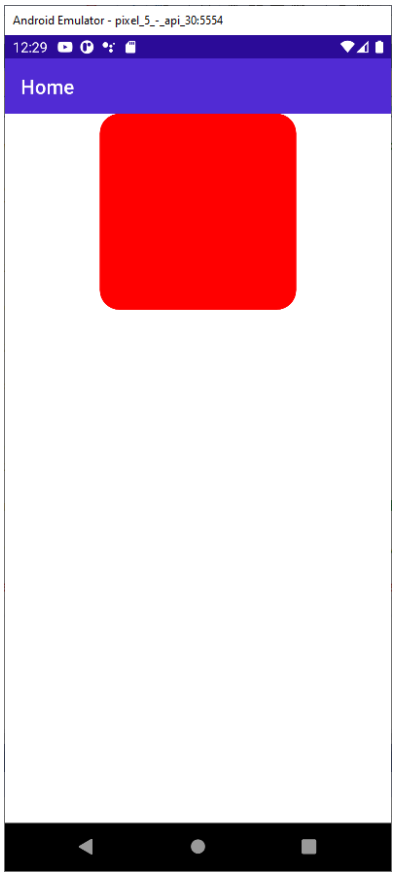

Figure 39: Displaying a BoxView

Introducing gesture recognizers

Views such as **`Image`** and **`Label`** do not include support for touch gestures natively, but sometimes you might want to allow users to tap a picture or text to perform an action, such as navigating to a page or website. The **`Microsoft.Maui.Controls`** namespace has classes that allow you to leverage *gesture recognizers* to add touch support to views that do not include it out of the box.

Note: The documentation for gesture recognizers can be found here.

Views expose a collection called **`GestureRecognizers`**, of type **`IList<GestureRecognizer>`**. Supported gesture recognizers are:

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

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

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

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

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

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

Displaying alerts

All platforms can show pop-up alerts with informative messages or receive user input with common choices such as OK or Cancel. Pages in .NET MAUI provide an asynchronous method called **DisplayAlert**, which is very easy to use.

For example, suppose you want to display a message when the user taps a button. The following code demonstrates how to accomplish this:

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

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

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

You are free to write whatever text you like for the OK and Cancel options, and IntelliSense helps you understand the order of these options in the parameter list. If the user selects OK, **DisplayAlert** returns **true**. Figure 40 shows an example of the alert.

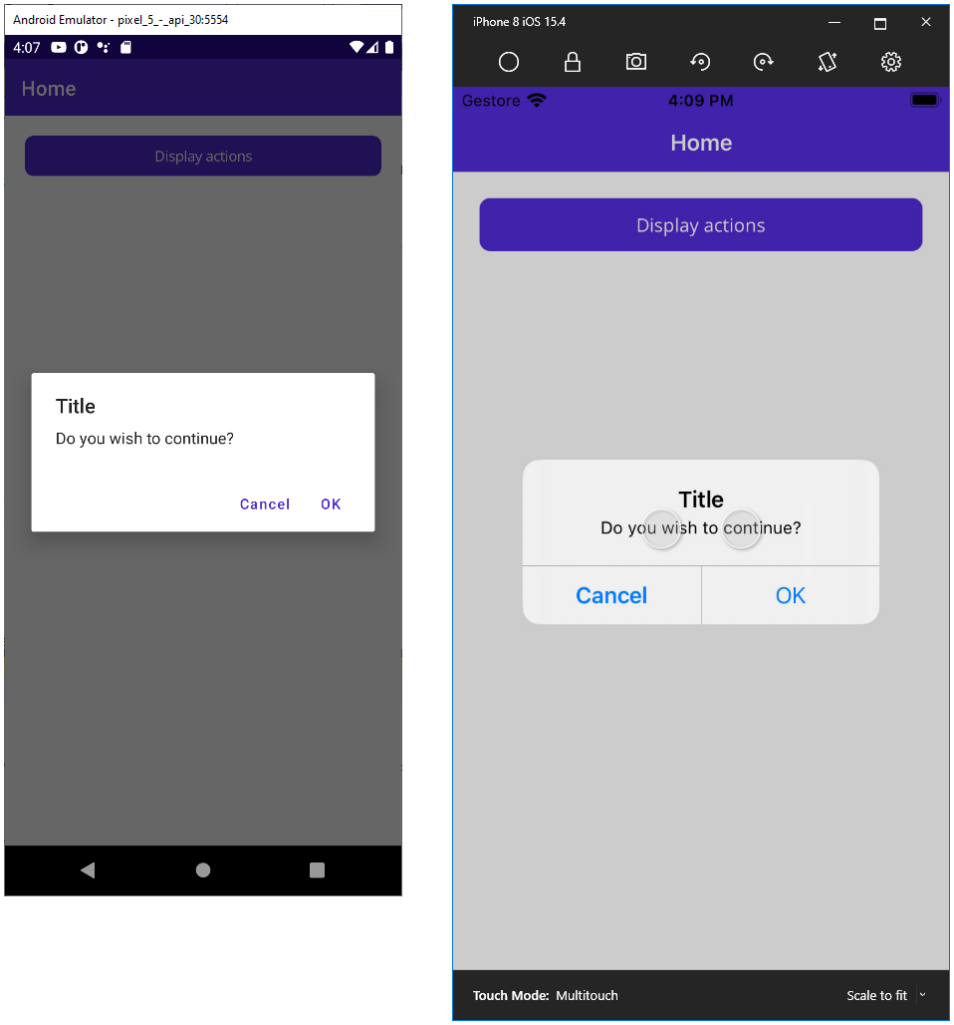

Figure 40: Displaying alerts

Displaying action sheets

An action sheet is a more sophisticated list of options that you can show inside a system alert and let the user make their choice. An action sheet is displayed via the **`DisplayActionSheet`** asynchronous method, like in the following example:

```
string result =
       await DisplayActionSheet("Select the destination", "Cancel",
       "Delete file", "Microsoft OneDrive", "Google Drive",
       "Apple iCloud", "Local device");
```

The preceding code simulates a list of potential targets for a file that is being created or copied. The first argument for the method is the alert title. The second argument is the string for the Cancel button. The third argument is known as the *destruction option* and allows for providing a command that represents a destructive action (in this case the file deletion).

This is an optional argument and can be assigned with **`null`** if you do not need a destruction option. Then you can specify as many options as you like, all of type **`string`**. The result of the method call is a string containing the text of the option that the user selected from the action sheet. Figure 41 shows how the sample action sheet looks.

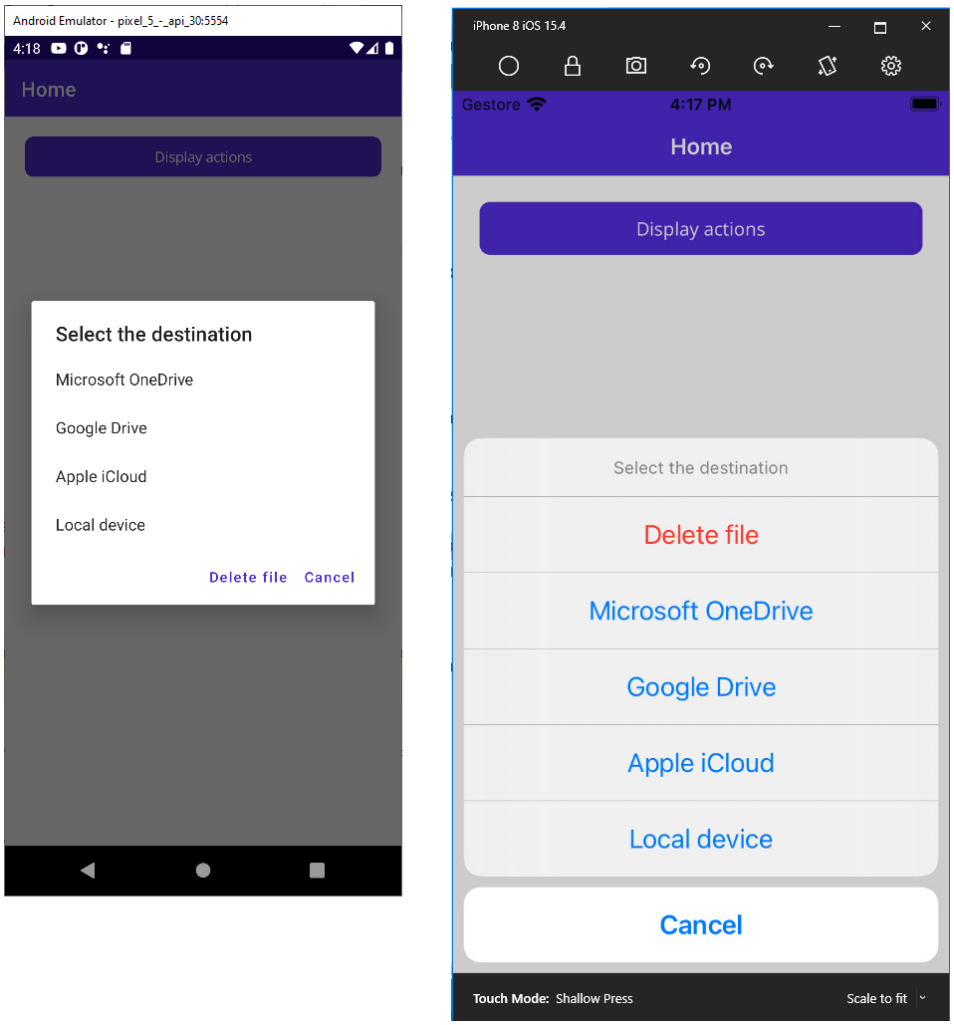

Figure 41: Displaying action sheets

Displaying action prompts

Sometimes you might need a quick and easy way to accept user input without implementing sophisticated visual elements. In this case, you can use a so-called action prompt. In its simplest form, you declare an action prompt as follows:

```
string promptResult =
       await DisplayPromptAsync("Welcome question",
       "Where did you hear about us from?");
```

The result is represented in Figure 42.

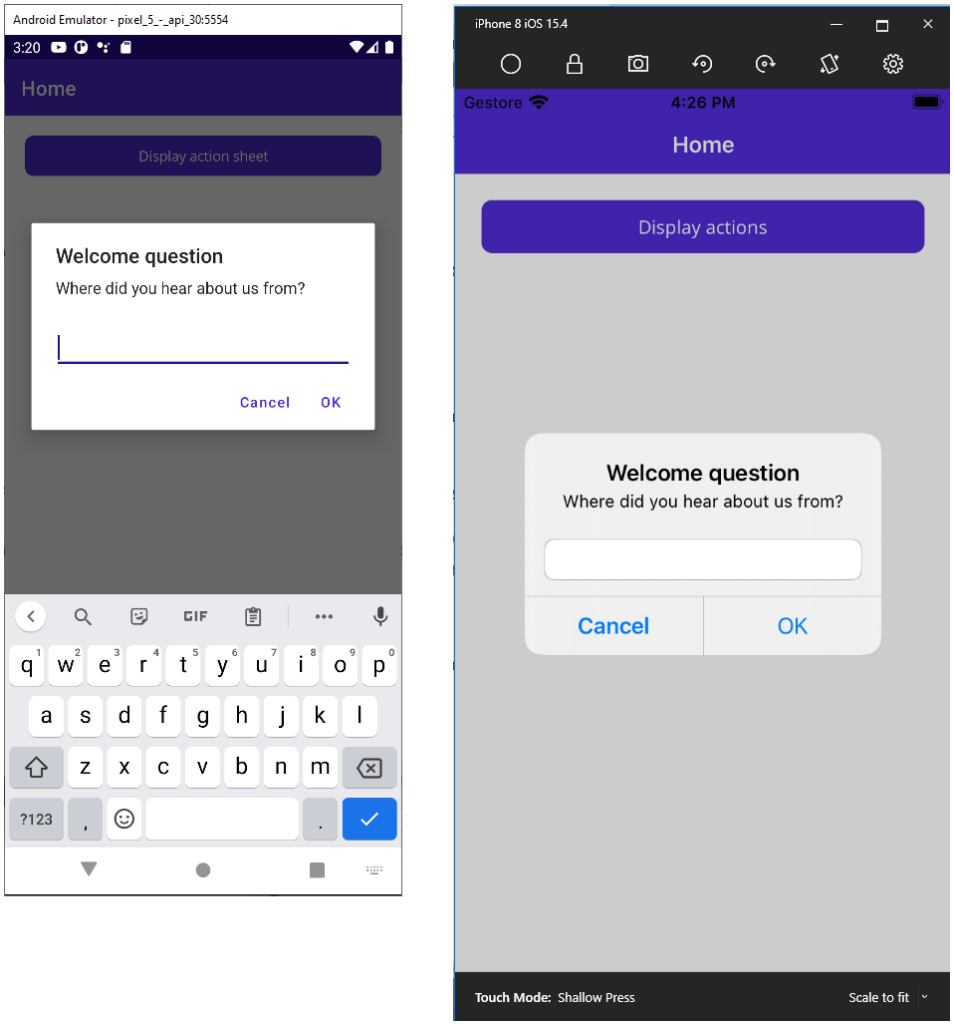

Figure 42: Displaying action prompts

The result of the method is a string. You can further customize the action prompt by specifying the text for the OK button (third argument), the Cancel button (fourth argument), the placeholder (fifth argument), the maximum length for the input text (sixth argument), the keyboard type (seventh argument), and an initial value (eighth argument).

The keyboard type is of type **`Keyboard`**, and you can specify the same values used with the **`Entry`** view.

Introducing the Visual State Manager

With the Visual State Manager, you can make changes to the user interface based on a view's state, such as **`Normal`**, **`Focused`**, or **`Disabled`**, all with declarative code. For example, you can use the Visual State Manager to change the color of a view depending on its state.

The following code snippet demonstrates how you can change the background color of an **`Entry`** view based on its state:

```
<Entry>
    <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="LightGray" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Disabled">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Gray" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

With this markup, the **`Entry`** will automatically change its background color when its state changes. In this case, you will need to set the **`Entry`**'s **`IsEnabled`** property as **`False`** in order to disable the view and trigger the **`Disabled`** state.

States must be grouped into objects called **`VisualStateGroup`**. Each state is represented by the **`VisualState`** object, where you add **`Setter`** specifications as you would do with styles, therefore providing the name of the property you want to change and its value.

Of course, you can specify multiple property setters. .NET MAUI, as well as all the other platforms based on XAML, defines a set of states called common states, such as **`Normal`**, **`Focused`**, and **`Disabled`** (see the **`VisualStateGroup`** with the **`CommonState`** name in the preceding code); these states are common to each view.

Other states might be available only to specific views and not to other views. The Visual State Manager provides an elegant and clean way to control the user interface behavior, all in your XAML code.

Chapter summary

This chapter introduced the concepts of view and common views in .NET MAUI, the building blocks for any user interface in your mobile apps. You have seen how to obtain user input with the **`Button`**, **`Entry`**, **`Editor`**, and **`SearchBar`** controls, how to display information with the **`Label`**, and how to use it in conjunction with other input views, such as **`Slider`**, **`Stepper`**, **`CheckBox`**, **`RadioButton`**, and **`Switch`**.

You have seen the **`DatePicker`** and **`TimePicker`** views and how they allow you to work with dates and time. You have learned how to display images with the **`Image`** view and videos with the **`MediaElement`**. You have seen how the **`WebView`** is used to show HTML content and how to inform users of the progress of long-running operations with **`ActivityIndicator`** and **`ProgressBar`**.

You have also learned how to add gesture support to views that do not include it out of the box, how to display alerts for informational purposes, and how to provide choices to users. Finally, you have seen how the Visual State Manager allows changing the user interface behavior based on a view's state with declarative code.

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

Chapter 6 Pages and Navigation

In the previous chapters, we went over the basics of layouts and views, which are the fundamental building blocks of the user interface of your applications. However, you only saw how to use layouts and views within a single page, while most real-world mobile apps are made of multiple pages.

The systems targeted by MAUI provide different pages that allow you to display content in several ways and to provide the best user experience possible based on the content you need to present. .NET MAUI provides unified page models you can use from your shared C# codebase that work cross-platform.

It also provides an easy-to-use navigation framework, which is the infrastructure you use to move between pages. In addition, you can leverage a simplified architecture in one place with the Shell. Pages, navigation, and the Shell are the last pieces of the user interface framework you need to know to build modern, native apps with .NET MAUI.

Note: In order to follow the examples in the first part of this chapter, either open the companion Chapter6_Navigation solution or create a new solution. If you choose the second option, every time a new page is discussed, just clean the content of the MainPage.xaml and MainPage.xaml.cs files (except for the constructor) and write the new code.

Introducing and creating pages

.NET MAUI provides many page objects that you can use to set up the user interfaces of your applications. Pages are root elements in the visual hierarchy, and each page allows you to add only one visual element, typically a root layout with other layouts and visual elements nested inside the root.

From a technical point of view, all the page objects in .NET MAUI derive from the abstract **`Page`** class, which provides the basic infrastructure of each page, including common properties such as **`Content`**. This is the most important property that you assign with the root visual element. Table 6 describes the available pages.

Table 6: Pages in .NET MAUI

Page Type	Description
`ContentPage`	Displays a single view object.
`TabbedPage`	Facilitates navigating among child pages using tabs.
`FlyoutPage`	Manages two separate panes and includes a flyout control.

Page Type	Description
NavigationPage	Provides the infrastructure for navigating among pages.

The next sections describe the available pages in more detail. Remember that Visual Studio provides item templates for different page types, so you can right-click the project in Solution Explorer, select **Add New Item**, and in the **Add New Item** dialog box, you can expand the .NET MAUI node to see templates for each page described in Table 6.

Single views with the ContentPage

The **ContentPage** object is the simplest page possible and allows for displaying a single visual element. You already looked at some examples of the **ContentPage** previously, but it is worth mentioning its **Title** property. This property is particularly useful when the **ContentPage** is used in pages with built-in navigation, such as **TabbedPage** and **FlyoutPage**, because it helps identify the active page.

The core of the **ContentPage** is the **Content** property, which you assign with the visual element you want to display. The visual element can be either a single control or a layout; the latter allows you to create complex visual hierarchies and real-world user interfaces. In XAML, the tag for the **Content** property can be omitted, which is a common practice (also notice **Title**):

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             Title="Main page"
             x:Class="App1.MainPage">

    <Label Text="A content page"/>
</ContentPage>
```

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

Splitting contents with the FlyoutPage

The **FlyoutPage** is a very important page since it allows you to split contents into two categories: generic and detail. The user interface provided by the **FlyoutPage** is very common in Android and iOS apps. It offers a flyout on the left (the generic part) that you can swipe to show and hide, and a second area on the right that displays more information (the detail part).

For example, a very common scenario for this kind of page is displaying a list of topics or settings in the flyout and the content for the selected topic or setting in the detail part. Both the flyout and the detail parts are represented by **ContentPage** objects. A typical declaration for a **FlyoutPage** looks like Code Listing 12.

Code Listing 12

```
<?xml version="1.0" encoding="utf-8" ?>
<FlyoutPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            Title="Main page"
            x:Class="App1.MainPage">

    <FlyoutPage.Flyout>
        <ContentPage Title="Main page">
            <Label Text="This is the Master" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </FlyoutPage.Flyout>
    <FlyoutPage.Detail>
        <ContentPage>
            <Label Text="This is the Detail" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </FlyoutPage.Detail>
</FlyoutPage>
```

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

Remember that assigning the **`Title`** property on the **`ContentPage`** that acts as the master is mandatory; otherwise, the runtime will throw an exception. The **`FlyoutPage`** object cannot work together with the Shell; this means that you need to change the following line in the **App.xaml.cs** file:

```
MainPage = new AppShell();
```

to the following:

```
MainPage = new MainPage();
```

This ensures that the **`FlyoutPage`** is used as the startup object instead of the Shell. You will need to make this change when using the **`TabbedPage`** as well, which is discussed in the next section.

Note: Every time you change the root page from `ContentPage` to another kind of page, such as `FlyoutPage`, you also need to change the inheritance in the code-behind. For example, if you open the C# MainPage.xaml.cs file, you will see that `MainPage` inherits from `ContentPage`, but in XAML you replaced this object with `FlyoutPage`. So, you also need to make `MainPage` inherit from `FlyoutPage`. If you

forget this, the compiler will report an error. This note is valid for the pages discussed in the next sections as well.

Figures 43 and 44 show the flyout and detail parts, respectively. You can swipe from the left to display the master flyout, and then swipe back to hide it. You can also control the flyout appearance programmatically by assigning the **`FlyoutLayoutBehavior`** with one of the following values:

- **`Default`**: The pages are displayed using each platform's default layout.
- **`Popover`**: The detail page covers the flyout page.
- **`Split`**: This equally splits the flyout page and the detail. The flyout is displayed on the left and the detail on the right.
- **`SplitOnLandscape`**: Similar to **`Split`**, but only applies when the device is in landscape orientation; otherwise, **`Default`** is assumed.
- **`SplitOnPortrait`**: Similar to **`Split`**, but only applies when the device is in portrait orientation; otherwise, **`Default`** is assumed.

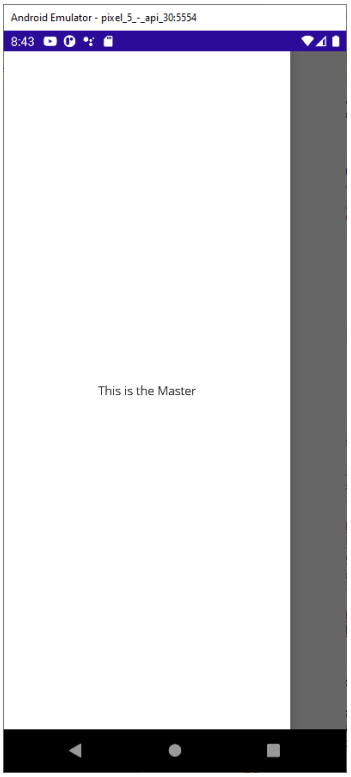

Figure 43: The Flyout of FlyoutPage

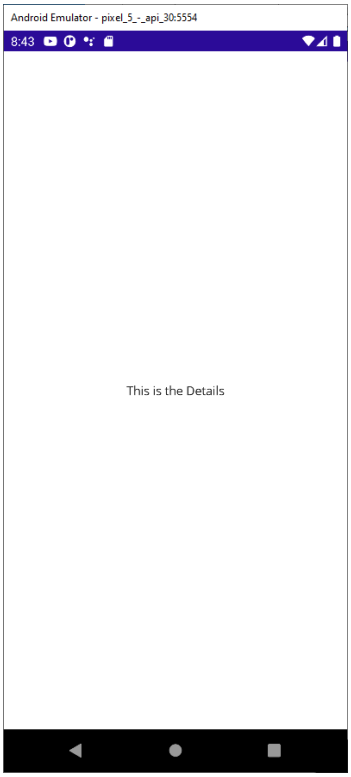

Figure 44: The Detail of FlyoutPage

Another interesting property is called **`IsPresented`**, which you assign with **`true`** (visible) or **`false`** (hidden). It's useful when the app is in landscape mode because the flyout is automatically opened by default. When not specified, **`IsPresented`** is **`false`**.

Displaying content within tabs with the TabbedPage

Sometimes you might need to categorize multiple pages by topic or by activity type. When you have a small amount of content, you can take advantage of the **TabbedPage**, which can group multiple **ContentPage** objects into tabs for easy navigation. The **TabbedPage** can be declared as shown in Code Listing 13.

Code Listing 13

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            Title="Main page"
            x:Class="App1.MainPage">

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

As you can see, you populate the **Children** collection with multiple **ContentPage** objects. Providing a **Title** to each **ContentPage** is of primary importance since the title's text is displayed in each tab, as demonstrated in Figure 45.

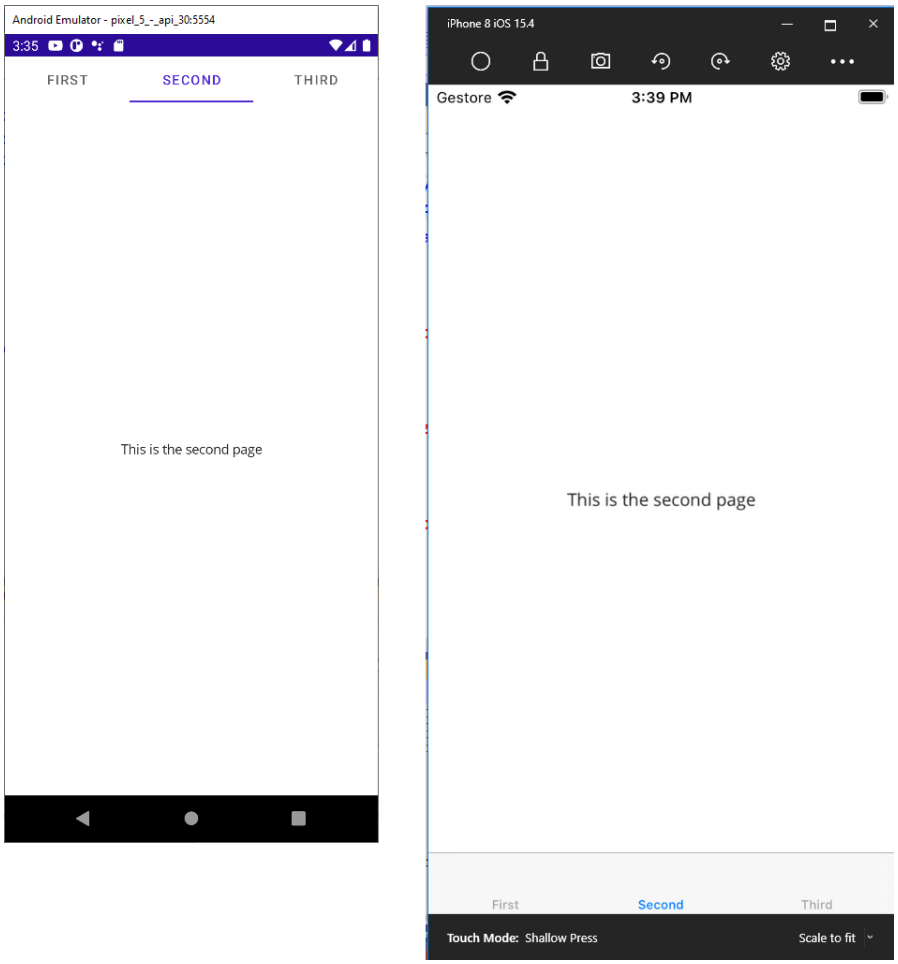

Figure 45: Displaying grouped contents with the TabbedPage

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

Navigating among pages

> ***Note: In the spirit of the* Succinctly *series, this section explains the most important concepts and topics of page navigation. However, there are tips and considerations that are specific to each platform that you must know when dealing with navigation in .NET MAUI. To learn more about them, see the official documentation.***

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

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

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

Wrapping a root page in a **NavigationPage** will not only enable the navigation stack, but will also enable the navigation bar on Android, iOS, and Windows. The navigation bar's text will be the value of the **Title** property of the current page object, represented by the **CurrentPage** read-only property.

Now suppose you added another page called **SecondaryPage.xaml** of type **ContentPage** to the project. The content of this page is not important at this point; just set its **Title** property with some text. If you want to navigate from the first page to the second page, use the **PushAsync** method as follows:

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

The **Navigation** property, exposed by each **Page** object, represents the navigation stack at the application level and provides methods for navigating between pages in a last-in, first-out (LIFO) approach. **PushAsync** navigates to the specified page instance. **PopAsync**, invoked from the current page, removes the current page from the stack and goes back to the previous page.

Similarly, **PushModalAsync** and **PopModalAsync** allow you to navigate between pages modally. The following lines of code demonstrate this:

```
// removes SecondaryPage from the stack and goes back to the previous page
await Navigation.PopAsync();

// displays the specified page as a modal page
await Navigation.PushModalAsync(new SecondaryPage());
await Navigation.PopModalAsync();
```

Figure 46 shows how the navigation bar appears on Android and iOS when navigating to another page.

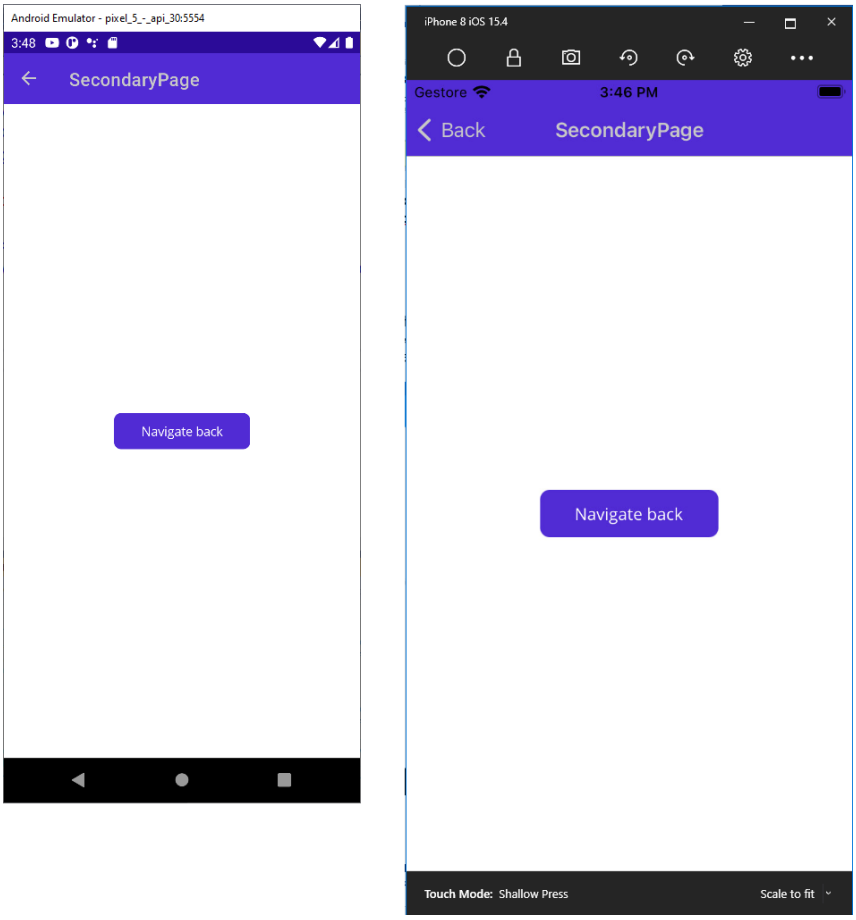

Figure 46: The navigation bar offered by the NavigationPage object

Users can just tap the Back button on the navigation bar to go back to the previous page. However, when you implement modal navigation, you cannot take advantage of the built-in navigation mechanism offered by the navigation bar, so it is your responsibility to implement code that allows going back to the previous page.

Modal navigation can be useful if you must be able to intercept a tap on the Back button on each platform. In fact, Android devices have a built-in hardware Back button that you can manage with events, but iOS does not. In iOS, you only have the Back button provided by the navigation bar, but this cannot be accessed by any events. So, in this case, modal navigation can be a good option to intercept user actions.

Passing objects between pages

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

Animating transitions between pages

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

Managing the page lifecycle

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

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

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

Actually, these events are not strictly related to navigation since they are available to any page, including individual pages. However, it is with navigation that they become very important, especially when you need to execute some code at specific moments in the page lifecycle.

For a better understanding of the flow, think of the page constructor: this is invoked the very first time a page is created. Then, **`OnAppearing`** is raised right before the page is rendered on screen. When the app navigates to another page, **`OnDisappearing`** is invoked, but this does not destroy the current page instance (this makes perfect sense).

When the app navigates back from the second page to the first page, the first page is not created again because it is still in the navigation stack, so its constructor will not be invoked, while **`OnAppearing`** will. So, within the **`OnAppearing`** method body, you can write code that will be executed every time the page is shown, while in the constructor, you can write code that will be executed only once.

Handling the hardware Back button

Android devices have a built-in hardware Back button for users instead of a Back button in the navigation bar. You can detect if the user presses the hardware Back button by handling the **`OnBackButtonPressed`** event as follows:

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

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

Simplified app architecture: The Shell

More often than not, mobile apps share several features like a navigation bar, a search bar, and a flyout menu that users can open by sliding from the left side of the screen. Implementing these features by yourself is not complex at all, but it requires a certain amount of time. Over the course of multiple projects, it can be a repetitive task.

In .NET MAUI, developers can leverage a root layout called Shell that is implemented by default when you create a new project inside the AppShell.xaml file. With the Shell, you can define the whole app architecture and hierarchy in one place and make use of a built-in navigation mechanism, search bar, and flyout menu.

In this chapter, you will learn the most common features of the Shell. However, this is a very sophisticated layout, so I suggest you bookmark the link to the official documentation page to keep as a reference for further study.

You will now see how to create an app with basic functionalities based on the sample project called **Shell**, located under the **Chapter6** folder of the companion repository. As you will see, the Shell definition resides in the AppShell.xaml file, which is the place where all the code discussed shortly needs to be placed.

Structure of the Shell

At its core, the Shell is a container of the following visual elements:

- **Flyout menu**: This is a side menu that can be shown or hidden with a swipe gesture and can contain other visual elements or, commonly, shortcuts to other pages. This is what Android users call a “hamburger” menu.
- **Tab bar**: This is usually placed at the bottom of the Shell and includes buttons with icons and text that users can tap to navigate to different pages.
- **Search bar**: This is used to search items in a list, typically by filtering a data-bound collection.

Visual elements in the Shell, such as flyout items and bar buttons, can be styled through resources. The flyout, tab bar, and search bar are independent from one another, so you could implement just one of them.

To implement a flyout, you just need to add **`FlyoutItem`** objects to the Shell. If you only want a tab bar instead of the flyout menu, you add a **`TabBar`** object to the Shell and then add a **`Tab`** object for each button you want to include. Then, you assign the **`FlyoutBehavior`** property of the Shell with **`Disabled`**. Both **`FlyoutItem`** and **`Tab`** need to be populated with an object of type **`ShellContent`**, which points to the page you want to open.

If you want to have both the flyout and the tab bar, you will nest a **Tab** object in a **FlyoutItem** object. If this all seems confusing, don't worry. You will see proper examples in the next sections.

Implementing the flyout menu

Suppose you want to create a flyout menu with three items, each pointing to a specific page. The code looks like the following:

```
<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>
```

For each **FlyoutItem** you can specify the **Title**, which is the text you see in the menu, and an icon (icon files must follow the rules of each operating system). The **ShellContent** object represents the page that is the target of the navigation.

The syntax based on the **ContentTemplate**, which receives a **DataTemplate** and the name of the target page as an argument, makes sure the instance of the target page is created only when required. You could also use the **Content** property instead of **ContentTemplate**, passing the name of the page directly, but in this case each page would be instantiated directly with potential performance overhead, so this is not recommended.

Data templates will be discussed in more detail in the next chapter as part of the data-binding topic. The result of this code is shown in Figure 47.

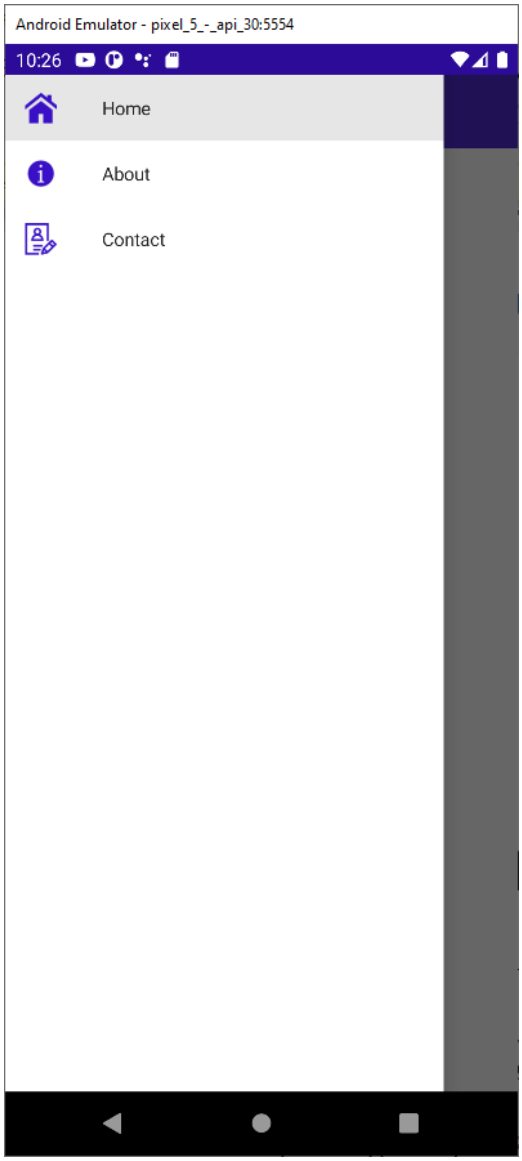

Figure 47: Flyout menu with the Shell

The flyout can be further customized via the following properties:

- **`FlyoutIcon`**: Sets the icon for the flyout.
- **`FlyoutHeader`**: Assigned with a view that appears at the top of the flyout. You can alternatively use the **`FlyoutHeaderTemplate`** property, which is populated with a **`DataTemplate`**.
- **`FlyoutBackgroundImage`**: Sets a background image for the flyout.
- **`FlyoutBackgroundImageAspect`**: Sets the aspect for the background image with the same values used for regular images (**`Fill`**, **`AspectFill`**, **`AspectFit`**).
- **`FlyoutIsPresented`**: Gets or sets the appearance status of the flyout. This can also be used in C# code to programmatically control the flyout appearance.

For a full list and description of the flyout properties, refer to the documentation.

Implementing the tab bar

Implementing the tab bar is accomplished by adding a **TabBar** object to the Shell, and then adding a **Tab** item for each navigation button you want. Continuing the previous example of three items, the code would look like this:

```
<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>
```

As you can see, each **Tab** has a title and icon, like the **FlyoutItem**. Navigation is again performed via the **ShellContent** object. Notice how you do not need to handle any events to perform navigation; everything is handled by the Shell. Figure 48 shows how the tab bar looks.

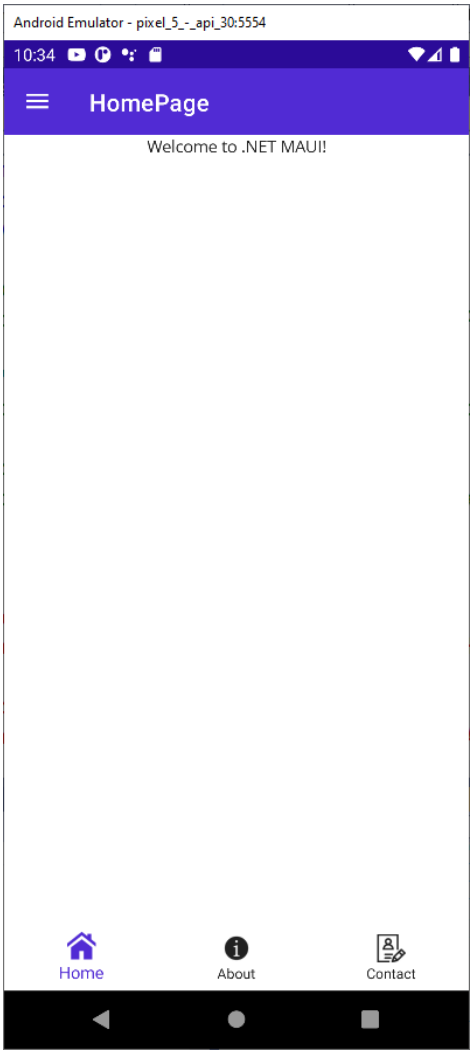

Figure 48: Tab bar with the Shell

Note: The Shell takes care of navigation between pages with its built-in mechanism, but you have deep control over it, and you can even manage navigation at runtime. This is the topic of so-called routing, and I recommend you have a look at the documentation for further study.

Implementing the flyout with tab bar

If you want to have the same elements in both the flyout and the tab bar, you can declare one **`FlyoutItem`** and add **`Tab`** objects inside it, like in 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 this case, you set the **FlyoutDisplayOptions** property with **AsMultipleItems**, which allows you to display several elements in a single flyout item. You can run the code and see how the app will show both the flyout with its items and the navigation bar.

Adding the search bar

Another piece of the Shell is the integrated search tool. The search tool is simply made of a class that derives from **SearchHandler**. Before implementing search, suppose your app displays a list of products that you want to filter as the user types. A product could be represented by the following **Product** class:

```
public class Product
{
    public string ProductName { get; set; }
    public int ProductQuantity { get; set; }
}
```

You need to then implement a collection of products. As you will learn in Chapter 7, XAML-based platforms work with the **ObservableCollection<T>** class.

You could then write the following code:

```
public class ProductCollection
{
    public ObservableCollection<Product> Products { get; set; }
    public ProductCollection()
    {
        var product1 = new Product { ProductName = "Bread", Quantity = 10 };
        var product2 = new Product { ProductName = "Wearables",
                                     Quantity = 15 };
        var product3 = new Product { ProductName = "Wine", Quantity = 5 };
        var product4 = new Product { ProductName = "Tomatoes",
                                     Quantity = 100 };

        Products = new ObservableCollection<Product>()
                       { product1, product2, product3, product4 };
    }
}
```

The **Products** property, of type **ObservableCollection<Product>**, will be the data source for both the list of products and the search UI. The next step is writing the search handler. Code Listing 14 shows an example.

Code Listing 14

```
    public class ItemsSearchHandler : SearchHandler
    {
        public ObservableCollection<Product> Products { get; set; }

        public ItemsSearchHandler()
        {
            Products = new ProductCollection().Products;
            ItemsSource = Products;
        }

        protected override void OnQueryChanged(string oldValue,
                  string newValue)
        {
            base.OnQueryChanged(oldValue, newValue);

            if (!string.IsNullOrWhiteSpace(newValue))
            {
                ItemsSource = Products.Where(p => p.ProductName
                .ToLower().Contains(newValue.ToLower()))
                .ToList();
            }

        }

        protected override async void OnItemSelected(object item)
        {
            base.OnItemSelected(item);

            await (App.Current.MainPage as Shell).

GoToAsync($"ObjectDetails?name={((Product)item).ProductName}");
        }
    }
```

The **ItemsSource** property of the search handler contains the filtered list of items and starts with the full data of the source collection. In the **OnQueryChanged** event handler, which is invoked when the user types in the search box, you will filter the collection that is bound to your page.

The **oldValue** and **newValue** string arguments of **OnQueryChanged** allow you to get the value in the search box before and after typing, respectively. In this example, a LINQ query filters the list of product names based on the user input.

Tip: The `OnQueryChanged` method is one of two key points for search implementation. This is the exact place where you implement your search logic, beyond a simple example like the previous one. The second key point is how you assign the `ItemsSource` property of the search handler. In this example, it has been done by assigning it with a sample list of products. In real-world scenarios, you might consider data binding or the assignment of lists populated with data coming from databases or view models.

The **`OnItemSelected`** event is raised when the user selects one of the items displayed in the search tool and allows for navigating to a specific page. The example assumes there is a page called **`ObjectDetails`**, to which the instance of the selected object is passed via query string as an argument, after conversion.

From the XAML point of view, you would add the following code inside a **`ContentPage`** definition (the companion solution does this in the HomePage.xaml page):

```
<Shell.SearchHandler>
    <local:ItemsSearchHandler Placeholder="Enter search term"
                              ShowsResults="true"
                              DisplayMemberName="ProductName" />
</Shell.SearchHandler>
```

The **`SearchHandler`** property is assigned with the **`SearchHandler`** implementation and allows you to specify placeholder text, whether the search box shows a preview of the results, and the name of the property from the bound collection that will be displayed (**`DisplayMemberName`**).

For a complete sample, you can add the following XAML code to display a list of products inside a **`CollectionView`** control:

```
<Grid Margin="20">
    <CollectionView ItemsSource="{Binding Products}"
                    x:Name="ProductsCollectionView">
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Label Text="{Binding ProductName}" />
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</Grid>
```

The **`CollectionView`** will be discussed in detail in the next chapter. For now, you just need to know that it is populated via data binding by assigning its **`ItemsSource`** property with an instance of the **`Products`** collection, and that the way it displays data is established through an **`ItemTemplate`** object.

The very last step for the example is creating an instance of the **`ProductCollection`** class and assigning it to the data source of the page:

```
public HomePage()
{
    InitializeComponent();
    var data = new ProductCollection();
    BindingContext = data;
}
```

Tip: As you will learn in the next chapter, the `BindingContext` property represents a page's data source.

Figure 49 shows what the search implementation looks like.

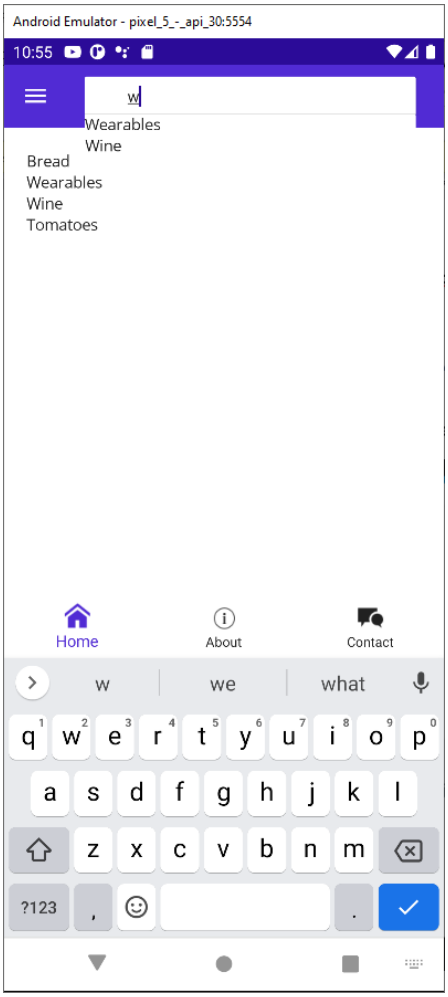

Figure 49: The search tool within the Shell

Styling the Shell

Elements in the Shell, such as the flyout and tab bar, can be customized with different colors and fonts. This is accomplished by defining specific resources. Resources in MAUI are discussed in the next chapter, so here you get a preview. For example, you could apply custom colors to the Shell and to tab bar elements with the following code:

```
<Shell.Resources>
    <ResourceDictionary>
        <Style x:Key="BaseStyle" TargetType="Element">
            <Setter Property="Shell.BackgroundColor" Value="LightGreen" />
            <Setter Property="Shell.ForegroundColor" Value="White" />
            <Setter Property="Shell.TitleColor" Value="White" />
            <Setter Property="Shell.DisabledColor" Value="#B4FFFFFF" />
            <Setter Property="Shell.UnselectedColor" Value="#95FFFFFF" />
            <Setter Property="Shell.TabBarBackgroundColor"
                    Value="LightBlue"/>
            <Setter Property="Shell.TabBarForegroundColor" Value="White"/>
            <Setter Property="Shell.TabBarUnselectedColor"
                    Value="#95FFFFFF"/>
            <Setter Property="Shell.TabBarTitleColor" Value="White"/>
        </Style>
  </ResourceDictionary>
</Shell.Resources>
```

Names for properties that accept styling are self-explanatory. You can also customize the appearance of the flyout and its elements. There are several additional ways to customize and configure the Shell, which are discussed in the documentation; I recommend you take a look for further study.

Note: There is much more to say about the Shell because it is a very sophisticated tool. This book could not cover topics such as custom navigation settings, custom renderers, or programmatic access to the Shell. If you consider implementing the Shell in your apps, I strongly recommend you bookmark the root of the official documentation, which contains everything you need to create full Shell experiences.

Chapter summary

This chapter has introduced the pages available in .NET MAUI, explaining how you can display single-view content with the **`ContentPage`** object, group content into tabs with the **`TabbedPage`**, and group content into two categories with the **`FlyoutPage`** object.

You have looked at how the **`NavigationPage`** object provides a built-in navigation framework that not only displays a navigation bar, but also allows for navigating between pages programmatically. Finally, you have been introducted to the Shell, an object that simplifies app infrastructure by including a flyout menu, a tab bar, and a search box in one place.

In the next chapter, you will look at information about two important and powerful features of .NET MAUI: resources and data binding.

Chapter 7 Resources and Data Binding

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

Working with resources

In XAML-based platforms, including .NET MAUI, resources are reusable pieces of information that you can apply to visual elements in the user interface. Typical XAML resources are styles, control templates, object references, and data templates. .NET MAUI supports styles and data templates, so these will be discussed in this chapter.

Declaring resources

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

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

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

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

Sometimes you might want to make resources available to the entire application. In this case, you can take advantage of the **App.xaml** file. The default code for this file is shown in Code Listing 15.

Code Listing 15

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:Resources"
             x:Class="Resources.App">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary
                 Source="Resources/Styles/Colors.xaml" />
                <ResourceDictionary
                 Source="Resources/Styles/Styles.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

As you can see, the autogenerated code of this file already contains an **`Application.Resources`** node with a nested **`ResourceDictionary`**. Resources you put inside this resource dictionary will be visible to any page, layout, and view in the application.

In addition, **`ResourceDictionary`** containers can be stored inside separate XAML files for easier reuse. When you want to link separate resource dictionaries, you add a collection called **`MergedDictionaries`** as you can see in the preceding code. Such a collection is populated with as many **`ResourceDictionary`** objects as separate XAML files you link, and the file name is specified by assigning the **`Source`** property.

When you create a new .NET MAUI project, Visual Studio generates two reusable resource dictionaries called Colors.xaml and Styles.xaml, both located inside the Resources\Styles subfolder. These files are very long, and they contain objects that you do not know yet, so their content will not be shown here. You can open them with Visual Studio and focus on the fact that the root node is a **`ResourceDictionary`** object. In the next section you will start learning about supported resources, and the content of the Colors.xaml and Style.xaml files will become clear.

Now that you have knowledge of where resources are declared and their scope, it is time to see how resources work, starting with styles. Other resources, such as data templates, will be discussed later in this chapter.

Introducing styles

When designing your user interface, you might have multiple views of the same type, and for each of them, you might need to assign the same properties with the same values. For example, you might have two buttons with the same width and height, or two or more labels with the same width, height, and font settings.

In such situations, instead of assigning the same properties many times, you can take advantage of styles. A style allows you to assign a set of properties to views of the same type. Styles must be defined inside a **ResourceDictionary** and must specify the type they are intended for and an identifier. The following code demonstrates how to define a style for **Label** views:

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

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

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

A style is applied by assigning the **Style** property on a view with an expression that encloses the **StaticResource** markup extension and the style identifier within curly braces. You can then assign the **Style** property on each view of that type instead of manually assigning the same properties every time.

With styles, XAML supports both **StaticResource** and **DynamicResource** markup extensions. In the first case, if a style changes, the target view will not be updated with the refreshed style. In the second case, the view will be updated, reflecting changes in the style.

Style inheritance

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

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

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

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

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

Implicit styling

A view's **Style** property allows you to assign a style defined inside resources. This allows you to selectively assign the style only to certain views of a given type. However, if you want the same style to be applied to all the views of the same type in the user interface, assigning the **Style** property to each view manually might be tedious.

In this case, you can take advantage of *implicit styling*. This feature allows you to automatically assign a style to all the views of the type specified with the **TargetType** property without the need to set the **Style** property. To accomplish this, you simply avoid assigning an identifier with **x:Key**, like in the following example:

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

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

Working with data binding

Data binding is a built-in mechanism that allows visual elements to communicate with data so that the user interface is automatically updated when data changes and vice versa. Data binding is available in all the most important development platforms, and .NET MAUI is no exception.

In fact, its data binding engine relies on the power of XAML, and the way it works is similar in all the XAML-based platforms. If you have previous experience with Xamarin.Forms, you'll notice no difference. .NET MAUI supports binding an object to visual elements, a collection to visual elements, and visual elements to other visual elements. This chapter describes the first two scenarios.

Because data binding is a very complex topic, the best way to start is with an example. Suppose you want to bind an instance of the following **Person** class to the user interface so that a communication flow is established between the object and views:

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

In the user interface, you will want to allow the user to enter their full name, date of birth, and address via an **Entry**, a **DatePicker**, and another **Entry**. In XAML, this can be accomplished with the code shown in Code Listing 16.

Code Listing 16

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBinding.SingleObjectPage"
             Title="SingleObjectPage">
    <VerticalStackLayout Padding="20">
        <Label Text="Name:" />
        <Entry Text="{Binding FullName}"/>

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

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

    </VerticalStackLayout>
</ContentPage>
```

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

Data binding can be of five types:

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

TwoWay and **OneWay** are the most-used modes, and in most cases, you do not need to specify the mode explicitly because MAUI automatically resolves the appropriate mode based on the view. For example, binding in the **Entry** control is **TwoWay** because this kind of view can be used to read and write data, whereas binding in the **Label** control is **OneWay** because this view can only read data.

However, with the **DatePicker**, you need to explicitly set the binding mode, so you use the following syntax:

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

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

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

Bindable properties will automatically update the value of the bound object's property and will automatically refresh their value in the user interface if the object is updated. However, this automatic refresh is possible only if the data-bound object implements the **`INotifyPropertyChanged`** interface, which allows an object to send change notifications.

Consequently, you must extend the **`Person`** class definition, as shown in Code Listing 17.

Code Listing 17

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

namespace App1
{
    public class Person : INotifyPropertyChanged
    {
        private string fullName;
        public string FullName
        {
            get
            {
                return fullName;
            }
            set
            {
                fullName = value;
                OnPropertyChanged();
            }
        }

        private DateTime dateOfBirth;

        public DateTime DateOfBirth
        {
            get
            {
                return dateOfBirth;
            }
            set
            {
                dateOfBirth = value;
                OnPropertyChanged();
```

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

        public event PropertyChangedEventHandler PropertyChanged;

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

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

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

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

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

Pages and layouts expose the **BindingContext** property (exactly like Xamarin.Forms), of type **object**, which represents the data source for the page or layout and is the same as **DataContext** in WPF or UWP.

Child views that are data-bound to an object's properties will search for an instance of the object in the **BindingContext** property value and bind to properties from this instance. In this case,

the **`Entry`** and the **`DatePicker`** will search for an object instance inside **`BindingContext`** and they will bind to properties from that instance.

Remember that XAML is case-sensitive, so binding to **`FullName`** is different from binding to **`Fullname`**. The runtime will throw an exception if you try to bind to a property that does not exist or has a different name.

If you now try to run the application, not only will data binding work, but the user interface will also be automatically updated if the data source changes. You may think of binding views to a single object instance, like in the previous example, as binding to a row in a database table.

IntelliSense support for data binding and resources

IntelliSense provides full support for binding expressions with markup extensions. For instance, suppose you have a **`Person`** class you want to use as the binding context for your UI and it is declared as a local resource.

IntelliSense will help you create the binding expression by showing the list of available resources. With a binding context declared in the resources, IntelliSense can help with creating binding expressions by showing a list of properties exposed by the bound object.

Figure 50 shows an example where you can also see the **`Text`** property of a **`Label`** being bound to the property **`FullName`**, which is defined in the **`Person`** class.

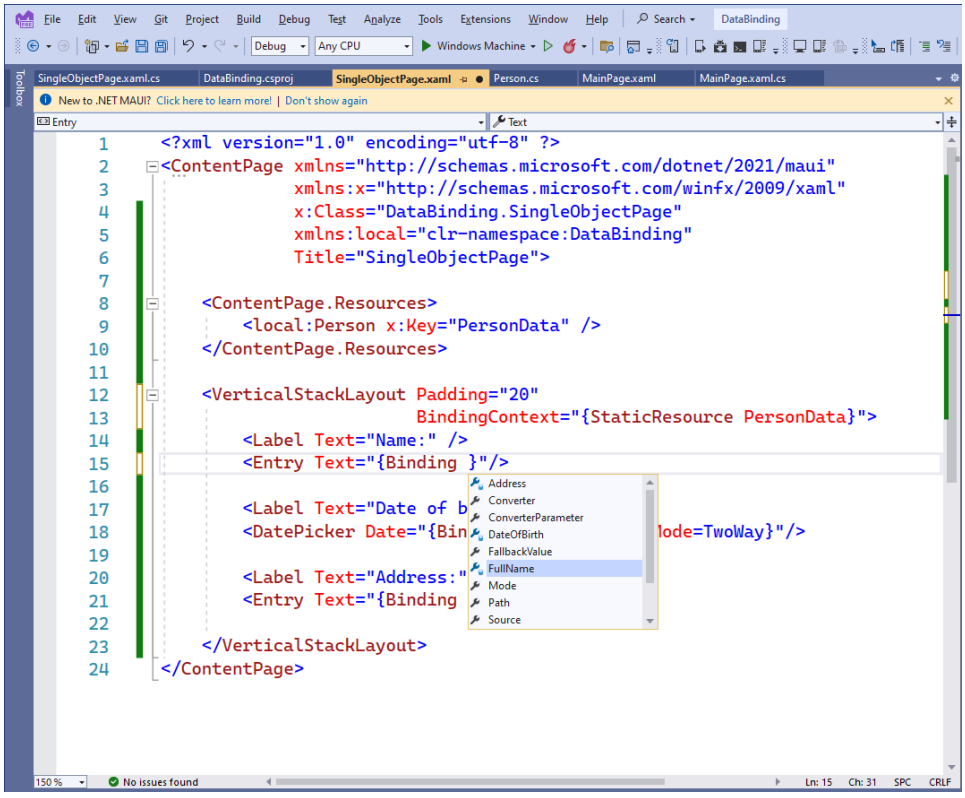

Figure 50: IntelliSense support for data-binding expressions

This is a big productivity feature that simplifies the way you create binding expressions.

Bindable spans

.NET MAUI offers the so-called bindable spans. The **`Span`** class inherits from **`BindableObject`**, which means that all its properties support data binding. The following snippet provides an example:

```
<Label
   HorizontalOptions="Center"
   VerticalOptions="Center">
    <Label.FormattedText>
        <FormattedString>
            <FormattedString.Spans>
                <Span FontSize="{Binding TitleSize}"
                      ForegroundColor="{Binding TitleColor}"
                      Text="{Binding Title}" />
                <Span FontSize="{Binding SubTitleSize}"
                      ForegroundColor="{Binding SubTitleColor}"
                      Text="{Binding SubTitle}" />
                <Span FontSize="{Binding AuthorSize}"
                      ForegroundColor="{Binding AuthorColor}"
                      Text="{Binding AuthorName}" />
            </FormattedString.Spans>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

With bindable spans, you can create formatted strings dynamically based on the data exposed by your view models.

Working with collections

Though working with a single object instance is a common scenario, another very common situation is working with collections that you display as lists in the user interface. .NET MAUI supports data binding over collections via the **`ObservableCollection<T>`** object. This collection works exactly like **`List<T>`**, but it also raises a change notification when items are added to or removed from the collection.

> ***Tip: To be precise, data binding in .NET MAUI works with every collection that implements the `IEnumerable<T>` interface. Using `ObservableCollection<T>` is generally preferred because of built-in change notification mechanisms, but other collections can be data-bound as well.***

Collections are very useful when you want to represent rows in a database table. For example, suppose you have the following collection of **`Person`** objects:

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

this.BindingContext = people;
```

The code assigns the collection to the **BindingContext** property of the root container, but at this point, you need a visual element that is capable of displaying the content of this collection. .NET MAUI offers two main views: the **ListView** and the **CollectionView**. In particular, the **ListView** is described here with the same level of detail because you might need to migrate existing code from Xamarin.Forms to MAUI.

> ***Note: If you have developed apps with Xamarin.Forms, you will notice how the `ListView` still relies on cell objects in .NET MAUI. The best approach would be to use the `CollectionView` whenever possible, but having the `ListView` as an option will certainly simplify the migration of your Xamarin.Forms projects to .NET MAUI.***

The **ListView** can receive the data source from either the **BindingContext** of its container or by assigning its **ItemsSource** property, and any object that implements the **IEnumerable** interface can be used with the **ListView**. You will typically assign **ItemsSource** directly if the data source for the **ListView** is not the same data source as for the other views in the page.

The problem to solve with the **ListView** is that it does not know how to present objects in a list. For example, think of a **People** collection that contains instances of the **Person** class, defined in Code Listing 18.

Code Listing 18

```
public class PeopleModel
{
    public ObservableCollection<Person> People { get; set; }

    public Person SelectedPerson { get; set; }

    public PeopleModel()
    {
        this.People = new ObservableCollection<Person>();
        Person person1 =
            new Person { FullName = "Alessandro",
                DateOfBirth = new DateTime(1977, 05, 10) };
        Person person2 =
            new Person { FullName = "James",
                DateOfBirth = new DateTime(1980, 02, 03) };
        Person person3 =
            new Person { FullName = "Graham",
                DateOfBirth = new DateTime(1982, 04, 06) };
```

```
        People.Add(person1);
        People.Add(person2);
        People.Add(person3);
    }
}
```

Each instance exposes the **`FullName`**, **`DateOfBirth`**, and **`Address`** properties, but the **`ListView`** does not know how to present these properties, so it is your job to explain to it how. This is accomplished with the *data templates*. A data template is a static set of views that are bound to properties in the object. It instructs the **`ListView`** on how to present items. Data templates in the **`ListView`** rely on the concept of cells.

Cells can display information in a specific way, as summarized in Table 7.

Table 7: Cells in .NET MAUI

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

Tip: Labels within cells are also bindable properties.

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

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

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

Generally, if the data source is assigned to the **`BindingContext`** property, the **`ItemsSource`** must be set with the **`{Binding}`** value, which means your data source is the same as that of your parent.

With this code, the **`ListView`** will display all the items in the bound collection, showing two cells for each item. However, each **`Person`** also exposes a property of type **`DateTime`**, and no cell is suitable for that. In such situations, you can create a custom cell using the **`ViewCell`**, as shown in Code Listing 19.

Code Listing 19

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page" Padding="20"
             x:Class="App1.MainPage">

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

As you can see, the **`ViewCell`** allows you to create custom and complex data templates, contained in the **`ViewCell.View`** property, so that you can display whatever kind of information

you need. Notice the **`HasUnevenRows`** property; if **`true`**, this dynamically resizes a cell's height based on its content.

> ***Tip: The `ListView` is a very powerful and versatile view and there is much more to it, such as interactivity, grouping and sorting, and customizations. I strongly recommend that you read the official documentation, which also includes some tips to improve performance.***

Figure 51 shows the result for the code provided in this section. Notice that the **`ListView`** includes built-in scrolling capability and should never be enclosed within a **`ScrollView`**.

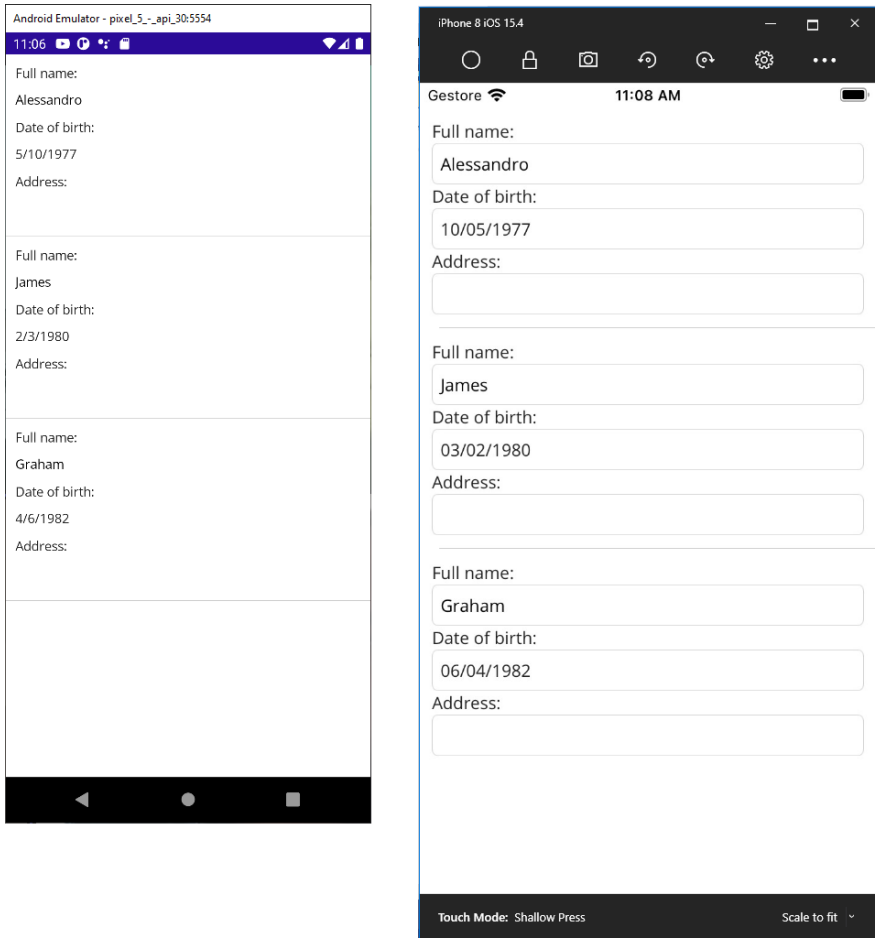

Figure 51: A data-bound ListView

A data template can be placed inside the page or app resources so that it becomes reusable. Then you assign the **`ItemTemplate`** property in the **`ListView`** definition with the **`StaticResource`** expression, as shown in Code Listing 20.

Code Listing 20

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
```

```
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:local="clr-namespace:App1"
            Title="Main page"
            x:Class="App1.MainPage">

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

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

You can also disable item selection with the **`SelectionMode = "None"`** property assignment. This can be useful when displaying read-only data.

Showing and selecting values with the Picker view

It is common to provide the user an option to select an item from a list of values, which can be accomplished with the **`Picker`** view. You can easily bind a **`List<T>`** or **`ObservableCollection<T>`** to its **`ItemsSource`** property and retrieve the selected item via its **`SelectedItem`** property. For example, suppose you have the following **`Fruit`** class:

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

In the user interface, suppose you want to ask the user to select a fruit from a list with the XAML shown in Code Listing 21.

Code Listing 21

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBinding.PickerPage"
             Title="PickerPage">
    <VerticalStackLayout>
        <Label Text="Select your favorite fruit:"/>
        <Picker x:Name="FruitPicker" ItemDisplayBinding="{Binding Name}"

              SelectedIndexChanged="FruitPicker_SelectedIndexChanged"/>

    </VerticalStackLayout>
</ContentPage>
```

As you can see, the **`Picker`** exposes the **`SelectedIndexChanged`** event, which is raised when the user selects an item. With the **`ItemDisplayBinding`**, you specify which property it needs to display from the bound object—in this case, the fruit name.

The **`ItemsSource`** property can, instead, be assigned either in XAML or in the code-behind. In this case, a collection can be assigned in C#, as demonstrated in Code Listing 22.

Code Listing 22

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

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

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

    private async void FruitPicker_SelectedIndexChanged(object sender,
                                                        EventArgs e)
    {
        var currentFruit = this.FruitPicker.SelectedItem as Fruit;
        if (currentFruit != null)
            await DisplayAlert("Selection",
            $"You selected {currentFruit.Name}", "OK");
    }
}
```

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

Figure 52 shows how the user can select an item from the Picker. An iOS version of the Picker is not shown because there is a known issue at the time of this writing with Visual Studio 2022.

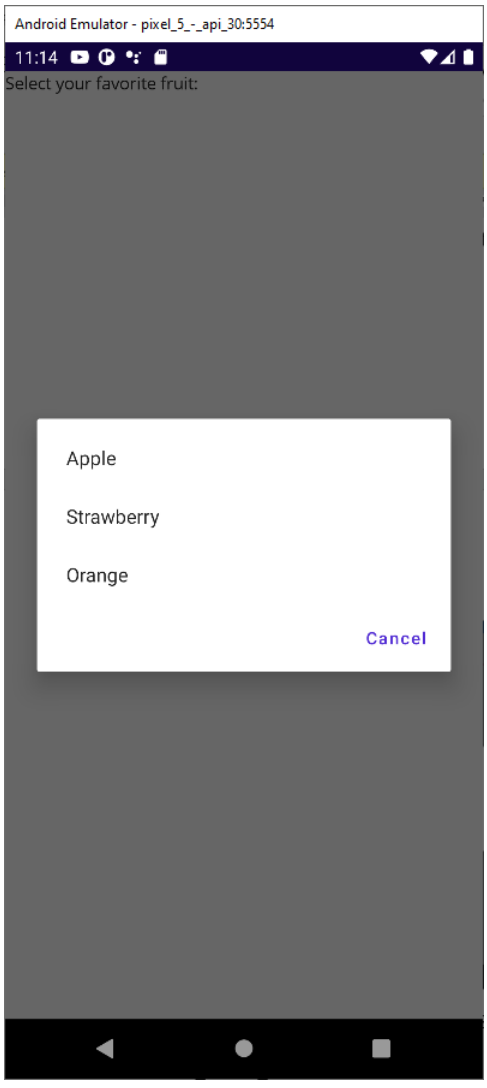

Figure 52: Selecting items with a Picker

Binding images

Displaying images in data-binding scenarios is very common, and .NET MAUI makes it easy to do. You simply need to bind the **`Image.Source`** property to an object of type **`ImageSource`**, or to a URL that can be represented by both a **`string`** and a **`Uri`**.

For example, suppose you have a class with a property that stores the URL of an image as follows:

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

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

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

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

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

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

Introducing value converters

The previous sentence mentioned a type converter that resolves specific types into the appropriate type for the **Image.Source** property. Such converters are available for many other views and types.

For example, if you bind an integer value to the **Text** property of an **Entry** view, such an integer is converted into a string by a XAML type converter. However, there are situations in which you might want to bind objects that XAML type converters cannot automatically convert into the type a view expects.

For example, you might want to bind a **Color** value to a **Label**'s **Text** property, which is not possible out of the box. In these cases, you can create *value converters*. A value converter is a class that implements the **IValueConverter** interface and exposes the **Convert** and **ConvertBack** methods. **Convert** translates the original type into a type that the view can receive, while **ConvertBack** does the opposite.

Code Listing 23 shows an example of a value converter that converts a string containing HTML markup into an object that can be bound to the **WebView** control. **ConvertBack** is not implemented because this value converter is supposed to be used in a read-only scenario, so a round-trip conversion is not required.

Code Listing 23

```
using System.Globalization;

namespace App1
{
    public class HtmlConverter : IValueConverter
```

```
    {
        public object Convert(object value, Type targetType,
               object parameter, CultureInfo culture)
        {
            try
            {
                var source = new HtmlWebViewSource();
                string originalValue = (string)value;

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

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

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

Code Listing 24 provides an example that also shows how to use the value converter.

Code Listing 24

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns=" http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

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

    <!-- Assumes you have a data-bound .NET object that exposes
```

```
    a property called HtmlContent -->
    <WebView Source="{Binding HtmlContent,
     Converter={StaticResource HtmlConverter}}"/>
</ContentPage>
```

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

Displaying lists with the CollectionView

Like Xamarin.Forms, .NET MAUI provides another view that allows you to display lists of data: the **`CollectionView`**. Its biggest benefits are a simpler API surface and no need for view cells, as you will see shortly. Whenever possible, use the **`CollectionView`** instead of the **`ListView`**.

Continuing the example shown previously, Code Listing 25 shows how you could bind a collection of **`Person`** objects to a **`CollectionView`**.

Code Listing 25

```
<CollectionView x:Name="PeopleList"
         ItemsSource="{Binding People}"
         SelectionMode="Single"
         SelectionChanged="PeopleList_SelectionChanged"
         VerticalScrollBarVisibility="Never"
         HorizontalScrollBarVisibility="Never">

    <CollectionView.ItemTemplate>
        <DataTemplate>
            <StackLayout Margin="10">
                <Label Text="Full name:"/>
                <Entry Text="{Binding FullName}"/>
                <Label Text="Date of birth:"/>
                <DatePicker Date="{Binding DateOfBirth,
                                    Mode=TwoWay}"/>
                <Label Text="Address:"/>
                <Entry Text="{Binding Address}"/>
            </StackLayout>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

You can control the visibility of the scroll bars directly, without custom renderers, by using the **`VerticalScrollBarVisibility`** and **`HorizontalScrollBarVisibility`** properties. The **`SelectionMode`** property allows for improved item selection with values like **`None`**, **`Single`**, and **`Multiple`**. The **`SelectionChanged`** event is fired when an item is selected.

Code Listing 26 shows how to work on the item selection, depending on the value of **`SelectionMode`**.

Code Listing 26

```
private void PeopleList_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    // In case of single selection
    var selectedPerson = this.PeopleList.SelectedItem as Person;

    // In case of multiselection
    var singlePerson = e.CurrentSelection.FirstOrDefault()
        as Person;

    var selectedObjects = e.CurrentSelection.Cast<Person>();
    foreach (var person in selectedObjects)
    {
        // Handle your object properties here...
    }
}
```

Figure 53 shows the result of the preceding code listings.

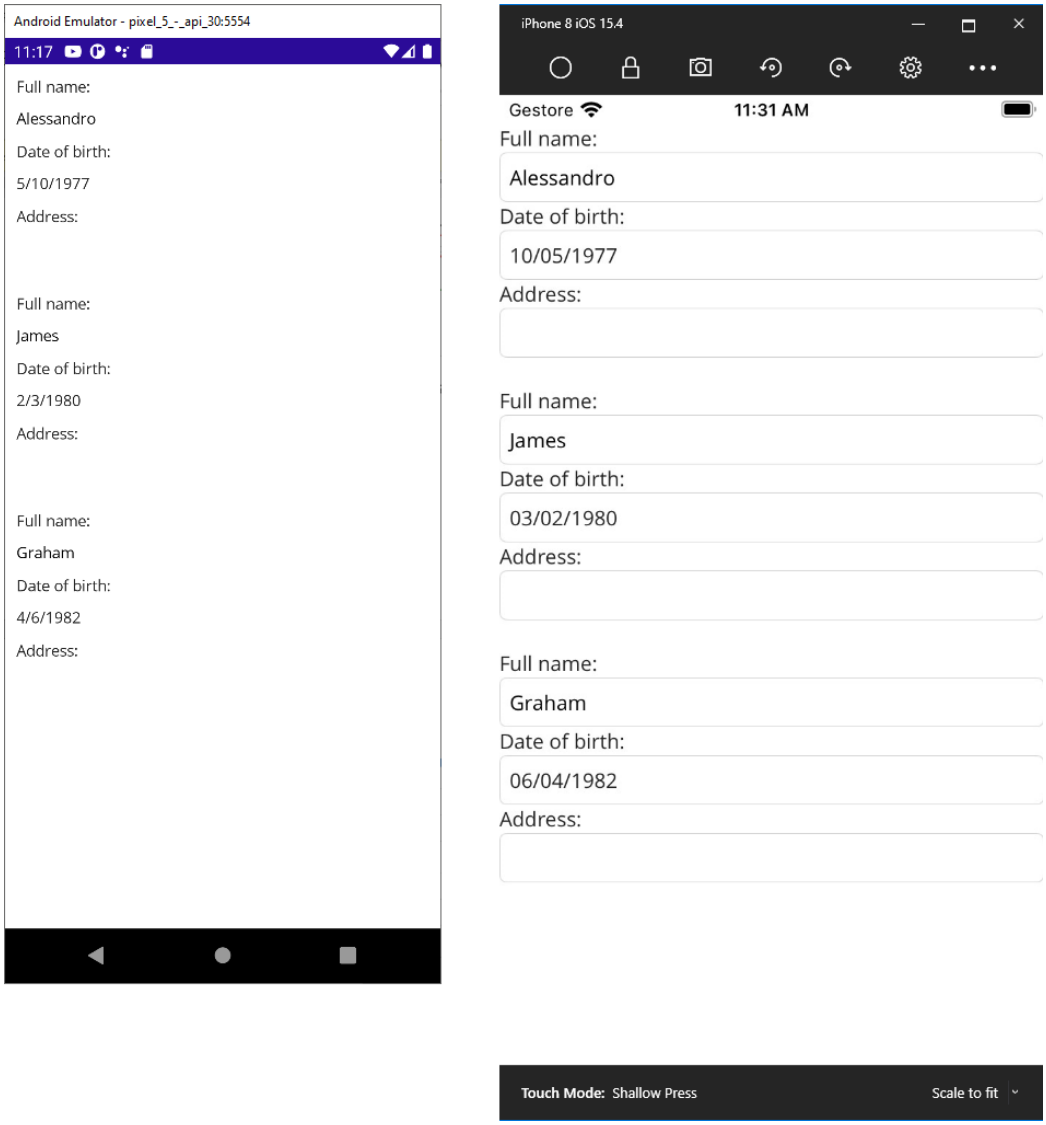

Figure 53: Displaying lists with the CollectionView

The **`CollectionView`** also makes it very simple to customize its layout. By default, the orientation is vertical, but you can use the **`LinearItemsLayout`** property to change the orientation to **`Horizontal`**, as follows:

```
<CollectionView.ItemsLayout>
    <LinearItemsLayout Orientation="Horizontal"/>
</CollectionView.ItemsLayout>
```

The **`CollectionView`** also supports a grid view layout, which can be accomplished with the **`GridItemsLayout`** property, as follows:

```
<CollectionView.ItemsLayout>
    <GridItemsLayout Orientation="Horizontal" Span="3"/>
</CollectionView.ItemsLayout>
```

The **Span** property indicates how many items are visible per line of the grid view. The **CollectionView** also has another amazing property called **EmptyView**, which you use to display a specific view in case the bound collection is empty (null or zero elements). For example, the following code demonstrates how to show a red label if the bound collection has no data:

```
<CollectionView.EmptyView>
    <Label Text="No data is available" TextColor="Red" FontSize="Large"/>
</CollectionView.EmptyView>
```

There is no similar option in the **ListView**, and it's your responsibility to implement and manage the empty data scenario. You can also display complex views with empty data by using the **EmptyViewTemplate** property instead, which works as shown in the following example:

```
<CollectionView.EmptyViewTemplate>
    <DataTemplate>
        <StackLayout Orientation="Vertical" Spacing="20">
            <Image Source="EmptyList.png" Aspect="Fill"/>
            <Label Text="No data is available" TextColor="Red"/>
        </StackLayout>
    </DataTemplate>
</CollectionView.EmptyViewTemplate>
```

In this code snippet, an image and a text message are displayed vertically. Figure 54 shows an example of an empty view.

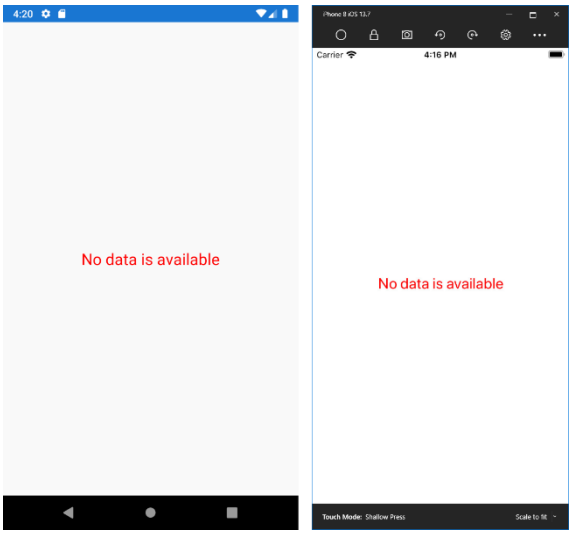

Figure 54: Default empty view

The **CollectionView** has no built-in support for the pull-to-refresh gesture; however, a view called **RefreshView** allows you to implement this common functionality. This view requires a bit of knowledge of the Model-View-ViewModel pattern, which is discussed later in this chapter.

Scrolling lists with the CarouselView

The **CarouselView** is a control that allows for scrolling lists. Typically, a **CarouselView** is used to scroll lists horizontally, but it also supports vertical orientation. When you declare a **CarouselView**, you will still define a **DataTemplate** like you would for the **CollectionView**, and you can also define the **EmptyView** and **EmptyViewTemplate** properties.

Code Listing 27 shows an example based on a collection of **Person** objects, just like the **CollectionView** example.

Code Listing 27

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBinding.CarouselViewPage" Padding="0,20,0,0"
             Title="CarouselViewPage">
    <VerticalStackLayout>
        <CarouselView x:Name="PeopleList" ItemsSource="{Binding People}"
                      CurrentItemChanged="PeopleList_CurrentItemChanged"
                      PositionChanged="PeopleList_PositionChanged"
                      CurrentItem="{Binding SelectedPerson}">
            <CarouselView.ItemsLayout>
                <LinearItemsLayout Orientation="Horizontal"
                         SnapPointsAlignment="Center"
                         SnapPointsType="Mandatory"/>
            </CarouselView.ItemsLayout>
            <CarouselView.ItemTemplate>
                <DataTemplate>
                    <StackLayout Margin="10">
                        <Label Text="Full name:"/>
                        <Entry Text="{Binding FullName}"/>
                        <Label Text="Date of birth:"/>
                        <DatePicker Date="{Binding DateOfBirth,
                                                   Mode=TwoWay}"/>
                        <Label Text="Address:"/>
                        <Entry Text="{Binding Address}"/>
                    </StackLayout>
                </DataTemplate>
            </CarouselView.ItemTemplate>
        </CarouselView>
    </VerticalStackLayout>
</ContentPage>
```

The **PositionChanged** event is fired when the control is scrolled. The **PositionChangedEventArgs** object provides the **CurrentPosition** and **PreviousPosition** properties, both of type **int**.

The **CurrentItemChanged** event is fired when the user taps on a displayed item. The **CurrentItemChangedEventArgs** object provides the **CurrentItem** and **PreviousItem** properties, both of type object. Notice how the **CurrentItem** property is data-bound to an individual instance of the **Person** class; this simplifies the way you get the instance of the currently selected object in the **CarouselView**.

In the layout definition, the **SnapPointsAlignment** property specifies how snap points are aligned with items, whereas the **SnapPointsType** property specifies the behavior of snap points when scrolling. Figure 55 shows an example of the **CarouselView** in action, with items scrolling horizontally.

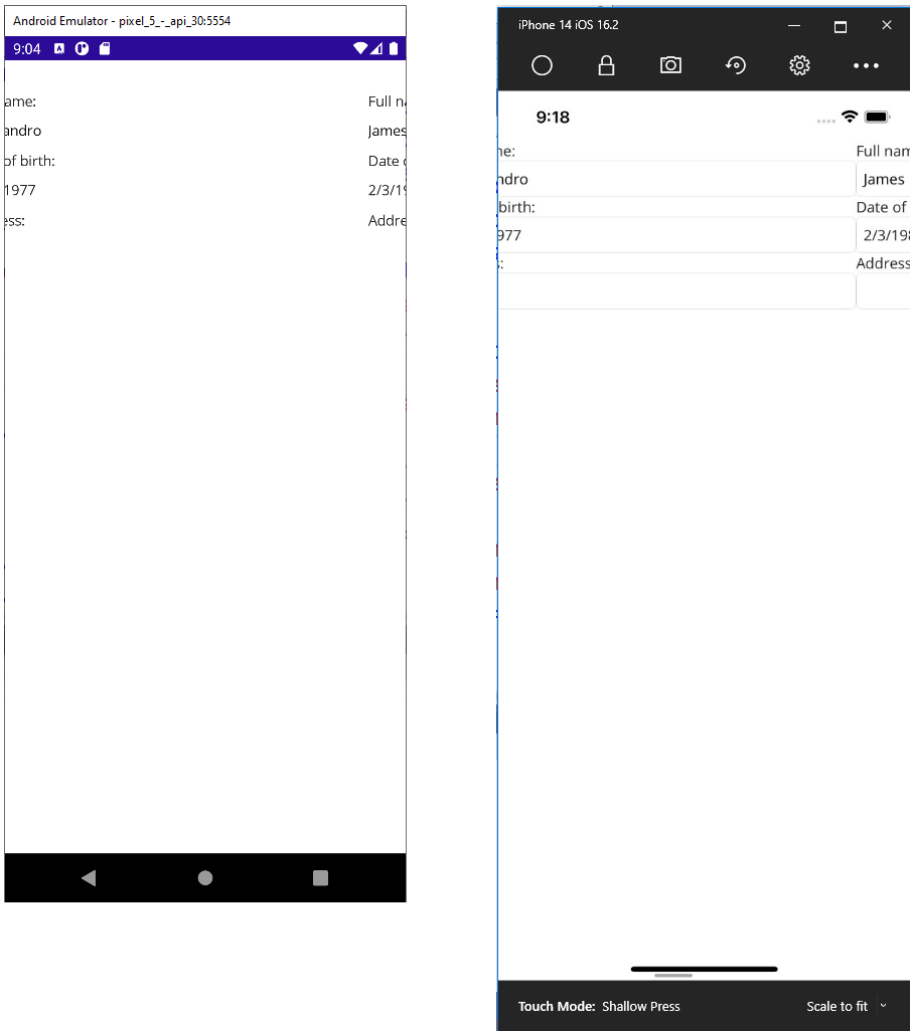

Figure 55: Scrolling lists with the CarouselView

Introducing the IndicatorView

The **IndicatorView** is a control that displays indicators representing the number of items and current position in a **CarouselView**. The latter has an **IndicatorView** property, which you assign with the name of the related **IndicatorView**, usually declared below the **CarouselView**.

Continuing the example of the previous section, you would extend the XAML as follows:

```
<VerticalStackLayout>
    <CarouselView IndicatorView="PersonIndicatorView">
      ...
    </CarouselView>
    <IndicatorView x:Name="PersonIndicatorView"
           IndicatorColor="LightGray"
           SelectedIndicatorColor="DarkGray"
           HorizontalOptions="Center" />
</VerticalStackLayout>
```

You will need to wrap both views into a layout to have them close to each other in the user interface. The **`IndicatorColor`** and **`SelectedIndicatorColor`** properties represent the color of the indicator in its normal and selected states.

You can also use the **`IndicatorsShape`** property and set it with **`Square`** (default) or **`Circle`** to define the appearance of the indicators. The **`IndicatorSize`** property allows you to customize the indicator size, but you can also provide a completely custom appearance using the **`IndicatorTemplate`** property, in which you can supply views to provide your own representation inside a **`DataTemplate`**.

Figure 56 shows an example of how the **`IndicatorView`** looks at the bottom of the image.

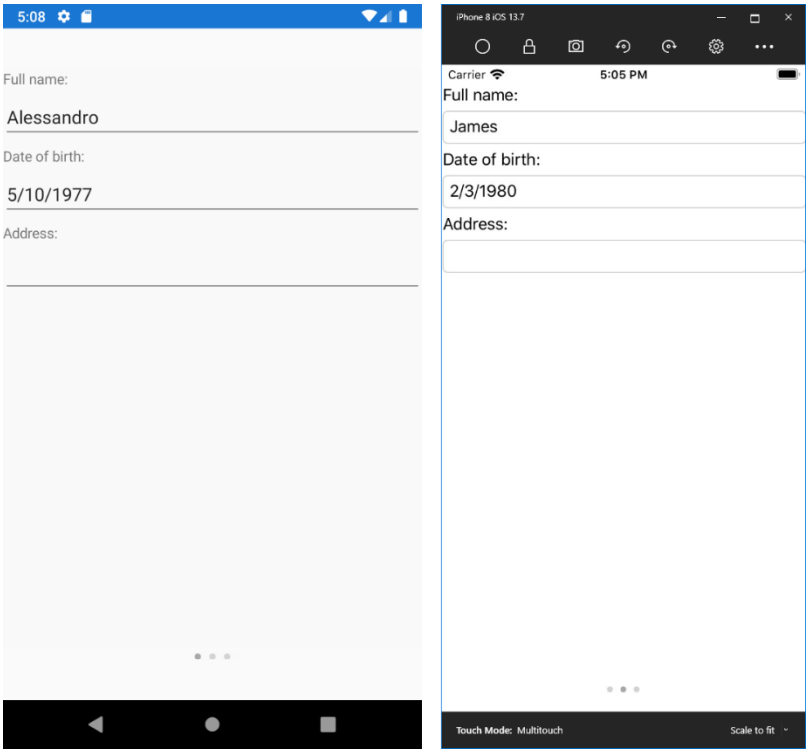

Figure 56: Assigning an IndicatorView to a CarouselView

Introducing Model-View-ViewModel

Model-View-ViewModel (MVVM) is an architectural pattern used in XAML-based platforms that allows for clean separation between the data (model), the logic (view model), and the user interface (view). With MVVM, pages only contain code related to the user interface, they strongly rely on data binding, and most of the work is done in the view model.

MVVM can be quite complex if you have never seen it before, so I will try to simplify the explanations as much as possible. The fundamental concepts that power MVVM are covered in the Commanding page of the official documentation.

Let's start with a simple example and a fresh .NET MAUI solution. Imagine you want to work with a list of **`Person`** objects. This is your model, and you can reuse the **`Person`** class from before. Add a new folder called **Model** to your project and add a new **Person.cs** class file to this folder, pasting the code of the **`Person`** class. Next, add a new folder called **ViewModel** to the project and add a new class file called **PersonViewModel.cs**.

Before writing the code for it, let's summarize some important considerations:

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

Loading, filtering, saving, and querying data are examples of actions a view model can execute against data. In a classic development approach, you would handle **`Clicked`** events on **`Button`** views and write the code that executes an action.

However, in MVVM, views should only contain code related to the user interface, not code that executes actions against data. In MVVM, view models expose the so-called commands. A command is a property of type **`ICommand`** that can be data-bound to views such as **`Button`**, **`SearchBar`**, **`CollectionView`**, **`ListView`**, and **`TapGestureRecognizer`** objects. In the UI, you bind a view to a command in the view model. In this way, the action is executed in the view model instead of in the view's code behind.

Code Listing 28 shows the **`PersonViewModel`** class definition.

Code Listing 28

```
using ModelViewViewModel.Model;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace ModelViewViewModel.ViewModel
{
    public class PersonViewModel : INotifyPropertyChanged
```

```
    {
        public ObservableCollection<Person> People { get; set; }

        private Person _selectedPerson;
        public Person SelectedPerson
        {
            get
            {
                return _selectedPerson;
            }
            set
            {
                _selectedPerson = value;
                OnPropertyChanged();
            }
        }

        public ICommand AddPersonCommand { get; set; }
        public ICommand DeletePersonCommand { get; set; }

        public event PropertyChangedEventHandler PropertyChanged;
        private void OnPropertyChanged([CallerMemberName] string
propertyName
                                       = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }

        private void LoadSampleData()
        {
            People = new ObservableCollection<Person>();

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

```
                new Person
                {
                    FullName = "Niklas",
                    Address = "Germany",
                    DateOfBirth = new DateTime(1980, 4, 2)
                };

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

        public PersonViewModel()
        {
            LoadSampleData();

            AddPersonCommand =
                new Command(() => People.Add(new Person()));

            DeletePersonCommand =
                new Command<Person>((person) => People.Remove(person));
        }
    }

}
```

The **People** and **SelectedPerson** properties expose a collection of **Person** objects and a single **Person**, respectively, and the latter will be bound to the **SelectedItem** property of a **CollectionListView**, as you will see shortly.

Notice how properties of type **ICommand** are assigned with instances of the **Command** class, to which you can pass an **Action** delegate via a lambda expression that executes the desired operation. The **Command** provides an out-of-the-box implementation of the **ICommand** interface, and its constructor can also receive a parameter, in which case you must use its generic overload (see the **DeletePerson** assignment).

In that case, the **Command** works with objects of type **Person** and the action is executed against the received object. Commands and other properties are data-bound to views in the user interface.

Note: Here you saw the most basic use of commands. However, commands also expose a `CanExecute` Boolean method that determines whether an action can be executed. Additionally, you can create custom commands that implement `ICommand` and must explicitly implement the `Execute` and `CanExecute` methods, where `Execute` is invoked to run an action. For further details, refer to the official documentation.

Now it is time to write the XAML code for the user interface. Code Listing 29 shows how to use the new **CollectionView** for this and how to bind two **Button** views to commands.

Code Listing 29

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ModelViewViewModel.MainPage">

    <VerticalStackLayout>
        <CollectionView x:Name="PeopleList"
                        ItemsSource="{Binding People}"
                        SelectionMode="Single"
                        SelectedItem="{Binding SelectedPerson}"
                        VerticalScrollBarVisibility="Never"
                        HorizontalScrollBarVisibility="Never">

            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <VerticalStackLayout Margin="10">
                        <Label Text="Full name:"/>
                        <Entry Text="{Binding FullName}"/>
                        <Label Text="Date of birth:"/>
                        <DatePicker Date="{Binding DateOfBirth,
                                  Mode=TwoWay}"/>
                        <Label Text="Address:"/>
                        <Entry Text="{Binding Address}"/>
                    </VerticalStackLayout>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
        <HorizontalStackLayout>
            <Button Text="Add" Command="{Binding AddPersonCommand}"/>
            <Button Text="Delete" Command="{Binding DeletePersonCommand}"
                    CommandParameter=
                    "{Binding Source={x:Reference PeopleList},
                    Path=SelectedItem}"/>
            <Button Text="Detail" Command="{Binding ViewPersonDetail}"/>
        </HorizontalStackLayout>
    </VerticalStackLayout>
</ContentPage>
```

Notice how:

- The **`CollectionView.ItemsSource`** property is bound to the **`People`** collection in the view model.
- The **`CollectionView.SelectedItem`** property is bound to the **`SelectedPerson`** property in the view model.
- The first **`Button`** is bound to the **`AddPerson`** command in the view model.
- The second **`Button`** is bound to the **`DeletePerson`** command, and it passes the selected **`Person`** object in the **`CollectionView`** with a special binding expression:

- **Source** represents the data source, in this case the **CollectionView**, referred to with **x:Reference**.
- **Path** points to the property in the source that exposes the object you want to pass to the command as a parameter (simply referred to as a *command parameter*).

The final step is to create an instance of the view model and assign it to the **BindingContext** of the page, which you can do in the page code-behind, as demonstrated in Code Listing 30.

Code Listing 30

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

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

        public MainPage()
        {
            InitializeComponent();

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

If you now run the application, you will see the list of **Person** objects, and you will be able to use the two buttons. The real benefit is that all the logic is in the view model. With this approach, if you change the logic in the properties or in the commands, you will not need to change the page code.

In Figure 57, you can see a new **Person** object added via command binding.

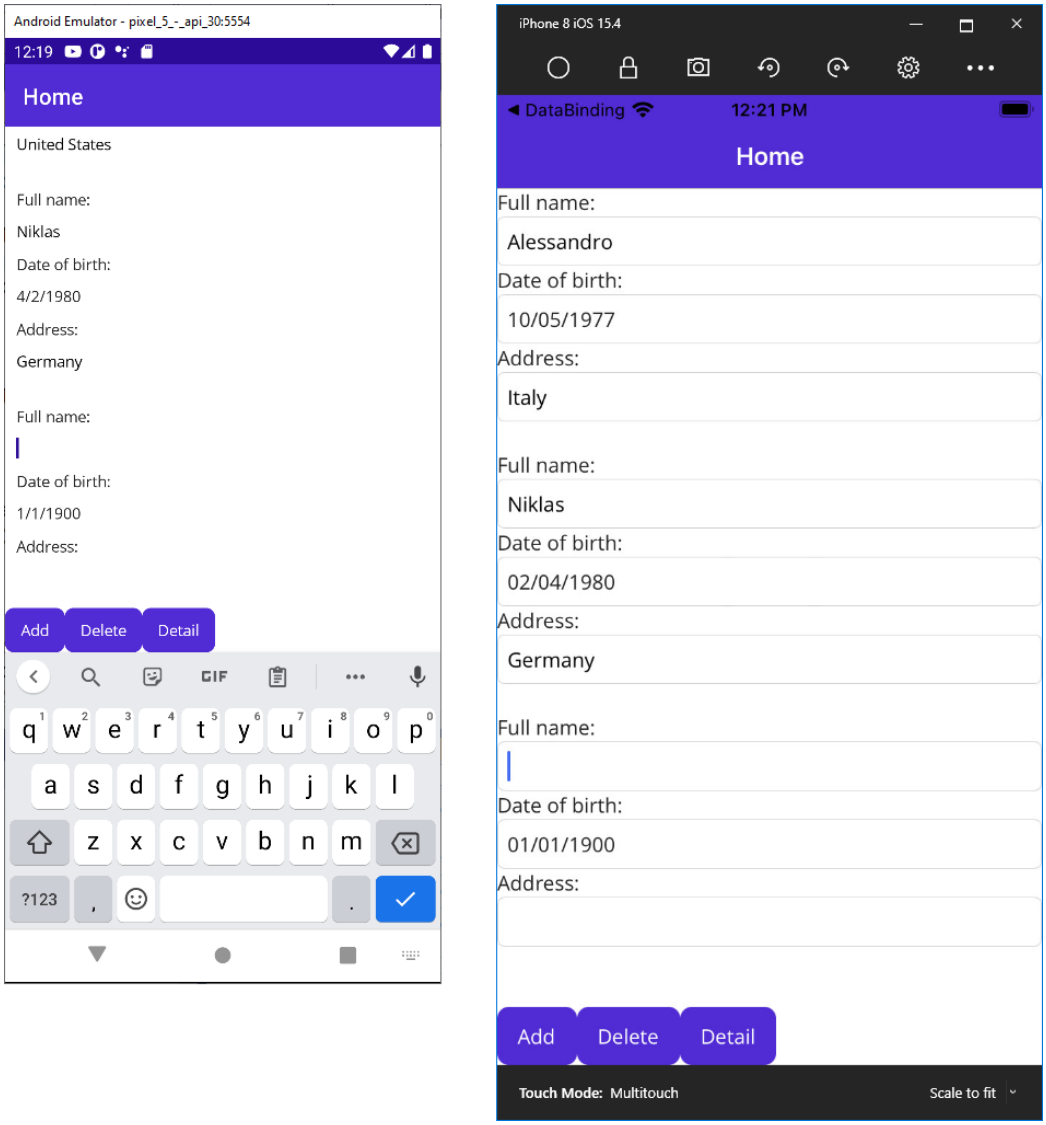

Figure 57: Showing a list of people and adding a new person with MVVM

MVVM is very powerful, but real-world implementations can be very complex. For example, if you want to navigate to another page and you have commands, the view model should contain code related to the user interface (launching a page) that does not adhere to the principles of MVVM.

Obviously, there are solutions to this problem that require further knowledge of the pattern, so I recommend you check out books and articles on the internet for further study. There's no need to reinvent the wheel; many robust and popular MVVM libraries already exist, and you might want to choose one from among the following:

- Prism
- MVVM Light Toolkit
- MvvmCross
- CommunityToolkit.Mvvm

All these alternatives are powerful enough to save you a lot of time.

Implementing pull-to-refresh: the RefreshView

The **CollectionView** provides many benefits due to a simplified API surface, but it lacks built-in support for the pull-to-refresh gesture. Fortunately, .NET MAUI provides the **RefreshView**, a container that allows for implementing the pull-to-refresh gesture where it isn't already available. This view is discussed here because, as you will see shortly, it requires knowledge of how commands work.

Tip: The `RefreshView` can be used with any views that implement scrolling, not just the `CollectionView`. For instance, a `ScrollView` with child elements can implement pull-to-refresh if put inside a `RefreshView`.

Let's start by adding the **RefreshView** in the XAML. Continuing the current code example, you will wrap the **CollectionView** inside a **RefreshView** as follows:

```
<VerticalStackLayout>
    <RefreshView RefreshColor="Teal"
                 IsRefreshing="{Binding IsRefreshing}"
                 Command="{Binding RefreshCommand}">
        <CollectionView x:Name="PeopleList" … >
        …
        </CollectionView>
    </RefreshView>
</VerticalStackLayout>
```

The **RefreshView** exposes a helpful property called **RefreshColor**, which allows you to change the color of the spinner that is displayed while the pull-to-refresh is active. The **IsRefreshing** property is bound to a **bool** property in the view model, which enables or disables the pull-to-refresh spinner when **true** or **false**, respectively.

The **Command** property is bound to a command object that performs the actual refresh operation. The **IsRefreshing** and **RefreshCommand** properties will be defined in the view model as follows:

```
private bool _isRefreshing;
public bool IsRefreshing
{
    get
    {
        return _isRefreshing;
    }
    set
    {
        _isRefreshing = value;
        OnPropertyChanged();
    }
```

```
}
public ICommand RefreshCommand { get; set; }
```

In the constructor of the view model, you can then assign the **RefreshCommand** with the action that will be performed to refresh the data, so add the following code:

```
RefreshCommand =
    new Command(async () =>
    {
        IsRefreshing = true;
        LoadSampleData();
        // Simulates a longer operation
        await Task.Delay(2000);
        IsRefreshing = false;
    }
);
```

The action does simple work: it assigns **IsRefreshing** with **true** in order to activate the pull-to-refresh user interface, it reloads the sample data, and then it assigns **IsRefreshing** with **false** when finished to disable the pull-to-refresh user interface.

Now you can run the sample, pull and release the list of data, and see how the gesture is implemented. Figure 58 shows an example.

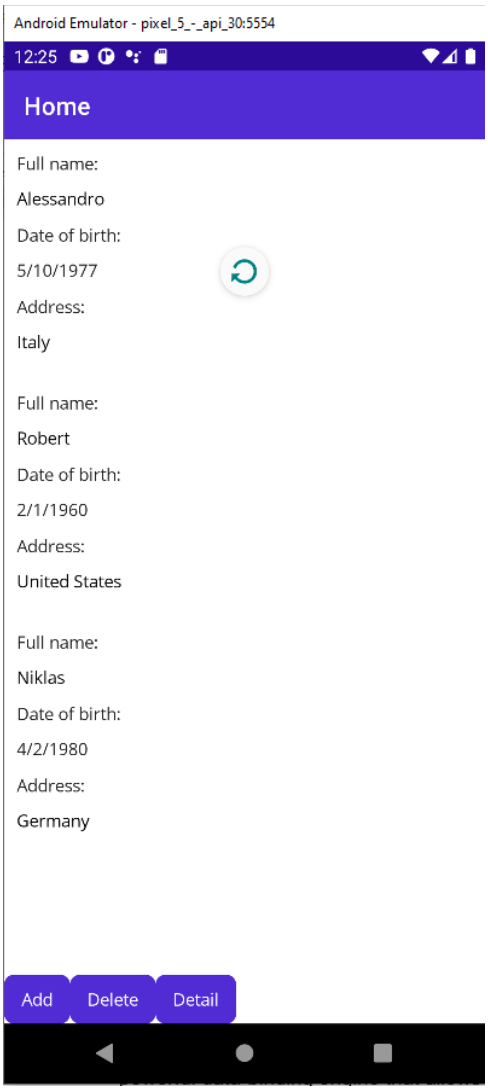

Figure 58: Pull-to-refresh with the RefreshView

Obviously, in real-world scenarios, your data will take a couple of seconds to reload, so you will not need to use the **`Task.Delay`** method to simulate a long-running operation.

Chapter summary

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

In this chapter, you have seen how to bind a single object to individual visual elements and collections of objects to **`ListView`** and **`CollectionView`**. You have seen how to define data

templates so that the **`ListView`** and the **`CollectionView`** can have knowledge of how items must be presented, and you have learned about value converters, special objects that help you when you want to bind objects of a type that is different from the type a view supports.

In the second part of the chapter, you have been introduced to the Model-View-ViewModel pattern, focusing on separating the logic from the UI and understanding new objects and concepts such as commands. Finally, you have seen how to quickly implement pull-to-refresh with the new **`RefreshView`**.

In the next chapter, you will walk through two exciting new features: brushes and shapes.

Chapter 8 Brushes and Shapes

If you have ever worked with professional designers on real-world mobile apps, you know how frustrating it is to say, “We can’t do this,” when they ask for shapes and gradient colors in the app design. There are options, like third-party libraries, but this creates a dependency, which is not always allowed by companies. Luckily, .NET MAUI offers brushes and shapes, providing a great solution to this kind of problem.

Understanding brushes

Brushes are types that expand the way you can add colors and gradients to your visual elements. Consider the following line, which declares a **`Button`** with a background color:

```
<Button Text="Tap here" BackgroundColor="LightBlue"/>
```

The **`BackgroundColor`** is of type **`Color`** and allows for assigning an individual color. You could also assign a property called **`Background`**, of type **`Brush`**. This is an abstract type definition for specialized types like **`SolidColorBrush`**, **`LinearGradientBrush`**, and **`RadialGradientBrush`**. Let’s discuss these in more detail.

Solid colors with SolidColorBrush

As the name implies, **`SolidColorBrush`** assigns a solid color to a property of type **`Brush`**. This can be done inline, like in the previous button declaration, which takes advantage of type converters. The value can be one of the literals displayed by IntelliSense or even a color in hexadecimal format (such as #FFFFFF).

`SolidColorBrush` can be assigned with two different syntaxes. The first one is an inline assignment, as follows:

```
<Frame CornerRadius="5" Background="Green"
       Margin="20" WidthRequest="150"
       HeightRequest="150">
</Frame>
```

The second syntax is the *property syntax*, with an extended form that will be very useful with gradients:

```
<Frame CornerRadius="5"
       Margin="20" WidthRequest="150"
       HeightRequest="150">
    <Frame.Background>
        <SolidColorBrush Color="Green"/>
    </Frame.Background>
</Frame>
```

Tip: In practice, using a solid color with a property of type `Color` or with a property of type `Brush` makes no difference. However, you might want to use properties of type `Brush` if you plan to replace a solid color brush value with a linear or radial gradient at runtime, since both of these derive from `Brush`, like `SolidColorBrush`.

Linear gradients with LinearGradientBrush

You can create linear gradients with the **`LinearGradientBrush`** object. Gradients can be horizontal, vertical, and diagonal. Linear gradients have two-dimensional coordinates, represented by the **`StartPoint`** and **`EndPoint`** properties on an axis where a value of 0,0 is the top-left corner of the view and 1,1 is the bottom-right corner of the view. The default value for **`StartPoint`** is **`0,0`** while the default value for **`EndPoint`** is **`1,1`**.

The following example draws a horizontal linear gradient as the background for a **`Grid`** view:

```
<Grid>
    <Grid.Background>
        <!-- StartPoint defaults to (0,0) -->
        <LinearGradientBrush EndPoint="1,0">
            <GradientStop Color="Blue"
              Offset="0.1" />
            <GradientStop Color="Violet"
              Offset="1.0" />
        </LinearGradientBrush>
    </Grid.Background>
</Grid>
```

Each **`GradientStop`** represents a color and its position in the gradient via the **`Offset`** property. Figure 59 shows what the gradient looks like.

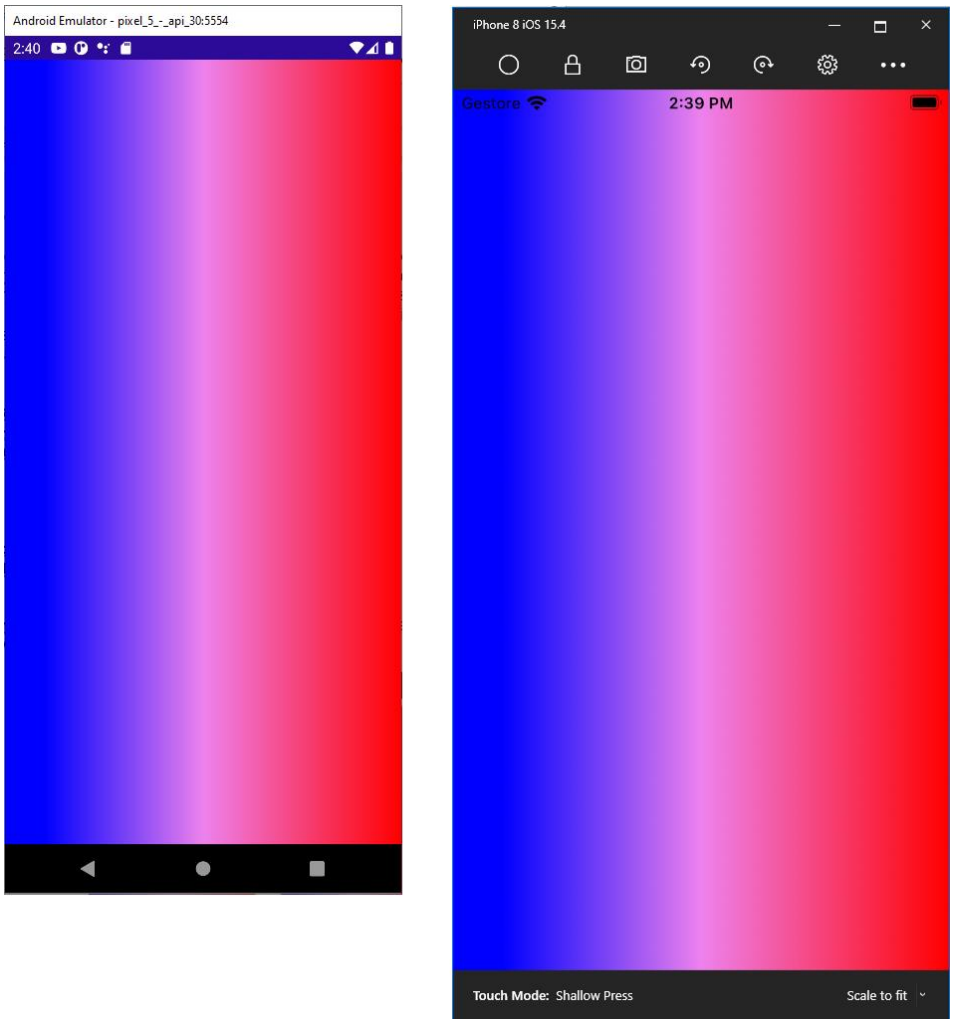

Figure 59: A horizontal linear gradient

You can change the orientation of the gradient to vertical by replacing the value of the **EndPoint** property as follows:

```
<LinearGradientBrush EndPoint="0,1">
```

A diagonal gradient can be created by simply removing **StartPoint** and **EndPoint** and using their default values of **0,0** and **1,1**, respectively. You can also add more than two colors. The following code demonstrates this:

```
<LinearGradientBrush EndPoint="1,0">
    <GradientStop Color="Blue"
      Offset="0.1" />
    <GradientStop Color="Violet"
      Offset="0.5" />
    <GradientStop Color="Red"
      Offset="1.0" />
</LinearGradientBrush>
```

Notice how the **Offset** property is used to determine the position of the colors; in this case, **Violet** is placed in the middle of the gradient in a scale between 0.1 and 1.

Circular gradients with RadialGradientBrush

You can create circular gradients using the **RadialGradientBrush** object. This object exposes the **Radius** property, of type **double**, which represents the radius of the circle for the gradient, and whose default value is **0.5**. It also exposes the **Center** property, of type **Point**, which represents the center of the circle in the gradient and whose default value is **0.5,0.5**.

In its simplest form, a **RadialGradientBrush** can be declared as follows:

```
<Grid Margin="10">
    <Grid.Background>
        <!-- Radius defaults to (0.5,0.5) -->
        <RadialGradientBrush>
            <GradientStop Color="Blue"
              Offset="0.1" />
            <GradientStop Color="Violet"
              Offset="1.0" />
        </RadialGradientBrush>
    </Grid.Background>
</Grid>
```

The resulting gradient looks like Figure 60.

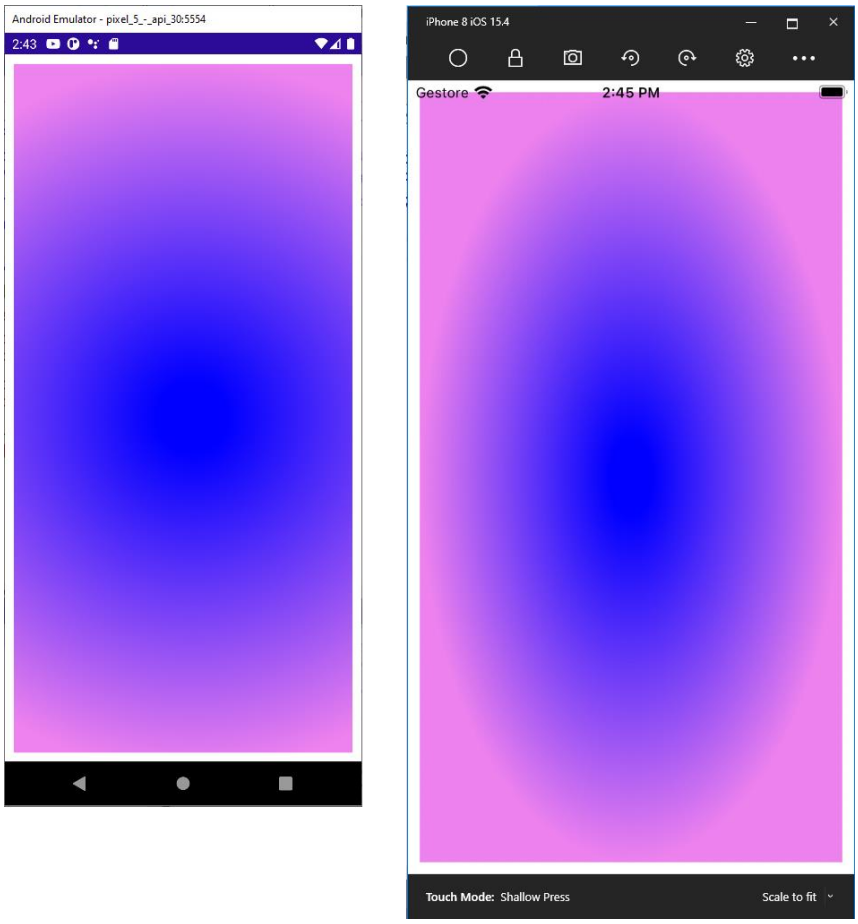

Figure 60: Drawing a circular gradient

You can move the center of the gradient with the **`Center`** property. The following value moves the center of the gradient to the top-left corner of the view:

```
<RadialGradientBrush Center="0.0,0.0">
```

The following assignment moves the center of the gradient to the bottom-right corner of the view:

```
<RadialGradientBrush Center="1.0,1.0">
```

In real-world development, you will probably use linear gradients more often, but radial gradients are certainly a very nice addition to the new drawing possibilities.

Drawing shapes

Shapes are views that derive from the **`Shape`** class, and .NET MAUI offers the following, each with a self-explanatory name: **`Ellipse`**, **`Rectangle`**, **`RoundRectangle`**, **`Line`**, and **`Polygon`**. In addition, a shape called **`Polyline`** allows for drawing custom polygons, and another shape called **`Path`** allows for drawing completely custom shapes.

All these shapes share some properties, and the most important are described in Table 8.

Table 8: Shape properties

Property	Type	Description
Aspect	**Stretch**	Defines how the shape fills its allocated space. Supported values are **None**, **Fill**, **Uniform**, and **UniformToFill**.
Fill	**Brush**	The brush used to fill the shape's interior.
Stroke	**Brush**	The brush used for the shape's outline.
StrokeThickness	**double**	The width of the shape's outline (default is 0).
StrokeDashArray	**DoubleCollection**	A collection of values that indicate the pattern of dashes and gaps used to outline a shape.
StrokeDashOffset	**double**	The distance between dashes.

Note: The examples provided shortly draw shapes' outlines for a better understanding, but this is totally optional.

Once you learn the properties listed in Table 8, you will see how shapes are very easy to handle.

Drawing ellipses and circles

The first shape we'll discuss is the **Ellipse**, which you use to draw ellipses and circles. The following code provides an example where the ellipse is filled with violet and the outline is green, with a thickness of **3**:

```
<Ellipse Fill="Violet" Stroke="Green" StrokeThickness="3"
 WidthRequest="250" HeightRequest="100" HorizontalOptions="Center"
 Margin="0,50,0,0"/>
```

The resulting shape is the one at the top in Figure 61.

Drawing rectangles

The second shape we'll discuss is the **Rectangle**. While keeping an eye on Table 8 for the properties used to draw dashes on the outline, review the following example that shows how you can use brushes to fill a shape:

```
<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="LightBlue" Offset="0"/>
            <GradientStop Color="Blue" Offset="0.5"/>
            <GradientStop Color="Violet" Offset="1"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

The resulting shape is the middle one in Figure 61. .NET MAUI also offers the **RoundRectangle** shape, which draws a rectangle with rounded corners by assigning its **CornerRadius** property.

Drawing lines

The next shape we'll discuss is the **Line**. Let's start with some code:

```
<Line X1="0" Y1="30" X2="250" Y2="20" StrokeLineCap="Round" Stroke="Violet"
 StrokeThickness="12" Margin="0,50,0,0" HorizontalOptions="Center"/>
```

X1 and **X2** represent the starting and ending points of the line on the x-axis. **Y1** and **Y2** represent the lowest and the highest points on the y-axis, respectively. The **StrokeLineCap** property, of type **PenCap**, describes a shape at the end of a line, and can be assigned with **Flat** (default), **Square**, or **Round**. **Flat** basically draws no shape, **Square** draws a rectangle with the same height and thickness of the line, and **Round** draws a semicircle with the diameter equal to the line's thickness.

Drawing polygons

Polygons are complex shapes, and MAUI provides the **Polygon** class to draw these. The following example draws a triangle filled with violet and outlined with green dashes:

```
<Polygon Points="50,20 80,60 20,60" Fill="Violet" Stroke="Green"
 StrokeThickness="4" StrokeDashArray="1,1" StrokeDashOffset="6"
 HorizontalOptions="Center"/>
```

The resulting shape is the bottom one in Figure 61.

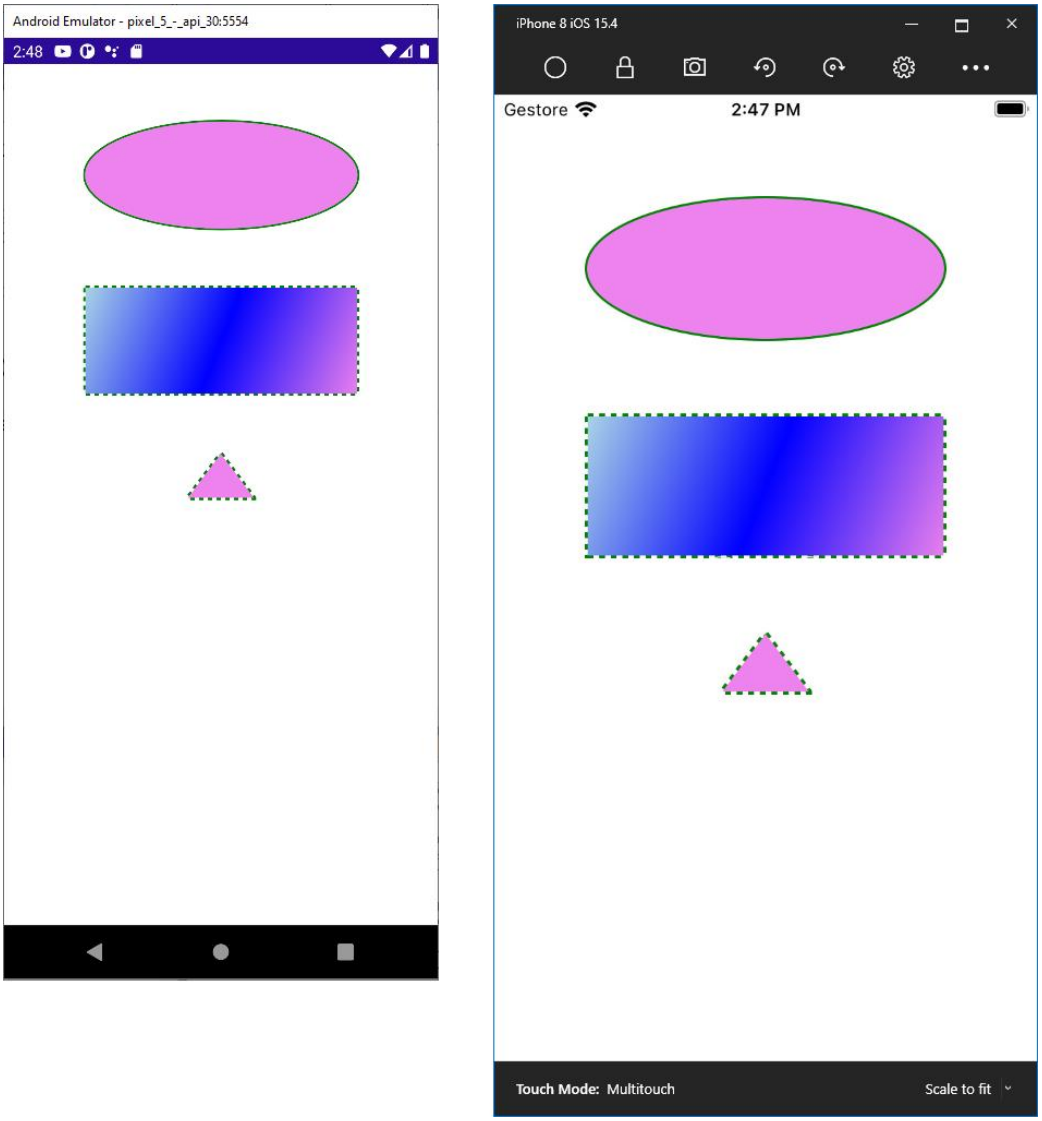

Figure 61: Drawing basic shapes

The key property in the **`Polygon`** shape is **`Points`**, which is a collection of **`Point`** objects, each representing the coordinates of a specific delimiter in the polygon. Because there's basically no limit to the collection of points, you can create very complex polygons.

Drawing complex shapes with the Polyline

The **`Polyline`** is a particular shape that allows for drawing a series of straight lines connected to one another, but the last line does not connect with the first point of the shape. Consider the following example:

```
<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" Stroke="DarkGreen" StrokeThickness="3"/>
```

Like the **`Polygon`**, the **`Polyline`** exposes a property called **`Points`**, which is a collection of **`Point`** objects, each representing the coordinates of a point. The resulting shape is shown in Figure 62.

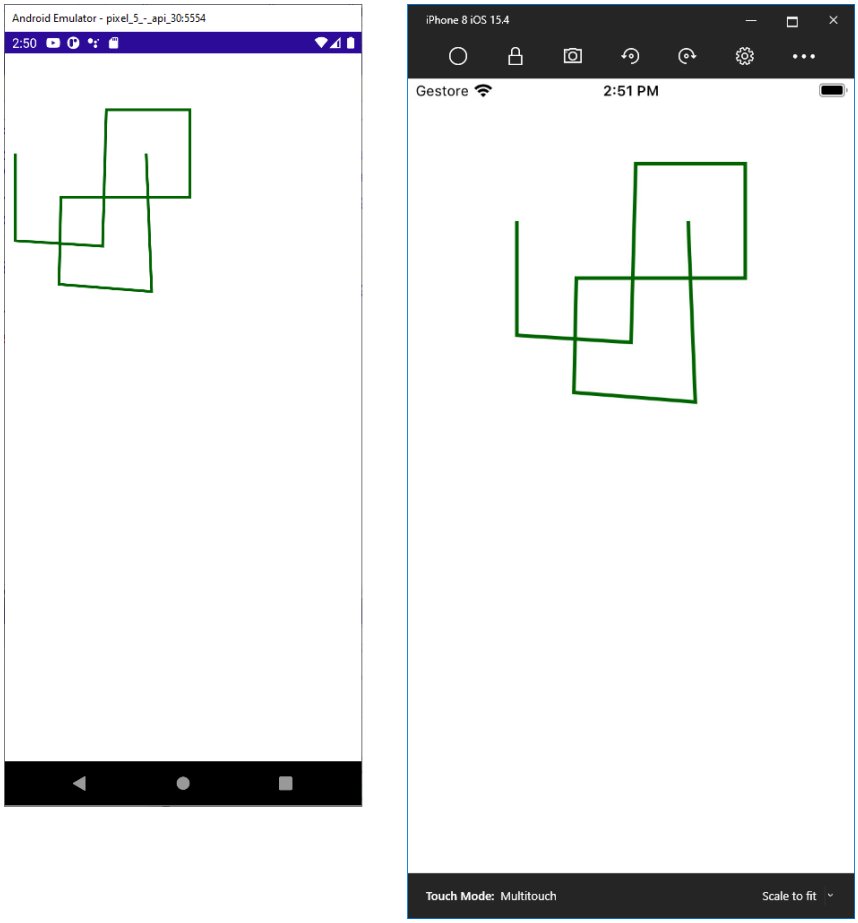

Figure 62: Drawing complex shapes

Note:** `Polyline` **exposes a property called** `FillRule`**, which determines how pieces of the shape should be filled based on the intersection of points. This is not a quick and easy topic, and it involves some math. For this reason, this property is not discussed in this ebook, but you can have a look at the documentation for further details.

Tips about geometries and paths

The new drawing possibilities are not limited to the basic shapes described in the previous sections. .NET MAUI also offers powerful 2D drawings with *geometries*, and it also provides the **`Path`** class that allows drawing curves and complex shapes using geometries. Both are very complex topics that cannot fit inside a book of the *Succinctly* series. For this reason, I recommend having a look at the official Geometries and Path documentation pages.

Chapter summary

This chapter introduced brushes and shapes. With brushes, you can assign solid colors (**`SolidColorBrush`**), linear gradients (**`LinearGradientBrush`**), and circular gradients (**`RadialGradientBrush`**) to properties of type **`Brush`**, like views' **`Background`** property. You can then draw shapes with the **`Ellipse`**, **`Rectangle`**, **`Line`**, **`Polygon`**, and **`Polyline`** objects, to which you can assign a brush to the **`Fill`** property as the shape color and the **`Stroke`** property as the outline color.

So far, you have seen what .NET MAUI has to offer cross-platform, but there is more power that allows you to use platform-specific APIs. This is the topic of the next chapter.

Chapter 9 Accessing Platform-Specific APIs

You have seen what .NET MAUI offers in terms of features that are available on each supported platform, walking through pages, layouts, and controls that expose properties and capabilities that will run on Android, iOS, Tizen, Mac Catalyst, and Windows.

Though this simplifies cross-platform development, it is not always enough to build real-world applications. In fact, especially with mobile apps, you will need to access sensors, the file system, the camera, and the network; send push notifications; and more. Each operating system manages these features with native APIs that cannot be shared across platforms and, therefore, that .NET MAUI cannot map into cross-platform objects.

Luckily, MAUI provides multiple ways to access platform-specific APIs that you can use to access practically everything on each platform. Thus, there is no limit to what you can do with this technology. To access platform features, you will need to write C# code in each platform folder. This is what this chapter explains, together with all the options you have to access iOS, Android, Tizen, Mac Catalyst, and Windows APIs from your shared codebase.

The DeviceInfo class and the OnPlatform method

The **`Microsoft.Maui.Devices`** namespace exposes an important class called **`DeviceInfo`**. This class allows you to detect the platform your app is running on, and the device type (tablet, phone, or desktop). This class is particularly useful when you need to adjust the user interface based on the platform.

The following code demonstrates how to take advantage of the **`DeviceInfo.Current`** property, which represents a singleton instance of the class, to detect the running platform and make UI-related decisions based on its value:

```
// Label1 is a Label view in the UI
if (DeviceInfo.Current.Platform == DevicePlatform.Android)
      Label1.Text = "I'm running on Android";
else if (DeviceInfo.Current.Platform == DevicePlatform.iOS)
    Label1.Text = "I'm running on iOS";
else if (DeviceInfo.Current.Platform == DevicePlatform.MacCatalyst)
    Label1.Text = "I'm running on Mac Catalyst";
else if (DeviceInfo.Current.Platform == DevicePlatform.Tizen)
    Label1.Text = "I'm running on Tizen";
else if (DeviceInfo.Current.Platform == DevicePlatform.WinUI)
    Label1.Text = "I'm running on WinUI";
```

`DevicePlatform` is of the same-named type and can be easily compared against specific constants—**`iOS`**, **`Android`**, **`macOS`**, **`Tizen`**, **`WinUI`**, **`MacCatalyst`**, **`tvOS`**, and **`UWP`**—that represent the supported platforms.

Tip: It is not possible to use a `switch` statement here, because this expects a constant value, but `DevicePlatform` is a type, not a constant.

The **DeviceInfo.Idiom** property allows you to determine if the current device the app is running on is a phone, tablet, or desktop computer, and returns one of the values from the **DeviceIdiom** enumeration:

```
if (DeviceInfo.Current.Idiom == DeviceIdiom.Tablet)
      Label1.Text = "I'm running on a tablet";
if (DeviceInfo.Current.Idiom == DeviceIdiom.Desktop)
    Label1.Text = "I'm running on a desktop computer";
if (DeviceInfo.Current.Idiom == DeviceIdiom.Watch)
    Label1.Text = "I'm running on a wearable device";
if (DeviceInfo.Current.Idiom == DeviceIdiom.Phone)
    Label1.Text = "I'm running on a phone";
if (DeviceInfo.Current.Idiom == DeviceIdiom.TV)
    Label1.Text = "I'm running on a tv";
if (DeviceInfo.Current.Idiom == DeviceIdiom.Unknown)
    Label1.Text = "Unknown device";
```

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

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             Padding="{OnPlatform '0',
                    iOS='0,20,0,0',
                    Android='0,10,0,0',
                    MacCatalyst='0,5,0,5'}"
             x:Class="NativeAccess.MainPage">
```

With the **OnPlatform** tag, you can specify a different property value based on the same values of the **DevicePlatform** structure. Differently from Xamarin.Forms, you no longer need to specify the **x:TypeArguments** attribute to represent the .NET type for the property, because this is inferred by Visual Studio. Similarly, you can also work with **OnIdiom** and the **DeviceIdiom** structure in XAML.

Tip: In iOS, it is best practice to set a page padding of 20 from the top, like in the previous snippet. If you don't do this, your page will overlap the system bar.

Finally, you can detect if your app is running on a simulator or a physical device. This is accomplished via the **DeviceType** structure, which exposes the **Virtual** and **Physical** values. Here is an example:

```
if (DeviceInfo.Current.DeviceType == DeviceType.Virtual)
    Label1.Text = "I'm on a simulator";
```

Understanding the device type is especially useful at debugging time.

Device-based localization

It is easy to implement UI localization based on the device settings by combining the **`FlowDirection`** property of each view with the **`Microsoft.Maui.ApplicationModel.AppInfo`** class, as follows:

```
FlowDirection = AppInfo.RequestedLayoutDirection.ToFlowDirection();
```

Suppose you are assigning **`FlowDirection`** on a page; this will allow you to apply UI localization settings to the whole page. The **`RequestedLayoutDirection`** property retrieves the settings for the device, but it needs the invocation of the **`ToFlowDirection`** method to convert the data into an object of type **`FlowDirection`**. In addition to XAML, you can also work with this property in C# code.

Invoking platform-specific code

Most of the time, apps need to offer interaction with the device hardware, sensors, system apps, and file system. Accessing these features from shared code is not possible because their APIs have unique implementations on each platform.

.NET MAUI provides a simple solution to this problem that relies on partial classes, each providing the platform-specific implementation of a task. This will be demonstrated shortly. For the sake of consistency, this section discusses the same example provided by the documentation, with some more considerations, including differences with Xamarin.Forms.

Implementing platform-specific code

Suppose you want to retrieve the current orientation of the device. This is handled from each system with unique APIs, so it cannot be done cross-platform directly.

The first thing you need to do is create a shared, partial class with members that will have a platform-specific implementation. Having that said, add to the project the code shown in Code Listing 31.

Code Listing 31

```
namespace NativeAccess
{
    public enum DeviceOrientation
    {
        Undefined,
        Landscape,
        Portrait
    }
```

```
    public partial class DeviceOrientationService
    {
        public partial DeviceOrientation GetOrientation();
    }
}
```

The key point is the **DeviceOrientationService** class, which is declared as partial. It defines a partial method called **GetOrientation**, so the compiler expects a full definition somewhere else. The platform-specific implementations will return a value from the custom **DeviceOrientation** enumeration.

The next step is adding platform-specific implementations of the class. When you need to implement platform-specific code, you add code files to the folder that represents the platform. For example, you will need to add a new class to the Android folder that looks like the code shown in Code Listing 32.

Code Listing 32

```
using Android.Content;
using Android.Runtime;
using Android.Views;

namespace NativeAccess
{
    public partial class DeviceOrientationService
    {
        public partial DeviceOrientation GetOrientation()
        {
            IWindowManager windowManager =
                Android.App.Application.Context
                .GetSystemService(Context.WindowService)
                .JavaCast<IWindowManager>();
            SurfaceOrientation orientation =
                windowManager.DefaultDisplay.Rotation;
            bool isLandscape = orientation ==
                SurfaceOrientation.Rotation90 ||
                orientation == SurfaceOrientation.Rotation270;
            return isLandscape ?
                DeviceOrientation.Landscape :
                DeviceOrientation.Portrait;
        }
    }
}
```

Tip: Partial classes must be defined inside the same namespace. For this reason, ensure that you change the namespace in the platform code to match the namespace in the shared partial class. In the current example, the namespace must be `NativeAccess`.

The focus here is more about the way invoking platform code works, rather than the description of every possible API. For now, what you need to know is that the code retrieves the instance of the current window converted into an object of type **`IWindowManger`**, and this allows for retrieving the rotation on Android devices via the **`SurfaceOrientation`** object.

Similarly, you will need to write code that retrieves the orientation on iOS. To accomplish this, add a new class to the iOS folder of the project and insert the code shown in Code Listing 33.

Code Listing 33

```
using UIKit;

namespace NativeAccess
{
    public partial class DeviceOrientationService
    {
        public partial DeviceOrientation GetOrientation()
        {
            UIInterfaceOrientation orientation =
                UIApplication.SharedApplication.StatusBarOrientation;
            bool isPortrait = orientation ==
                UIInterfaceOrientation.Portrait ||
                orientation == UIInterfaceOrientation.PortraitUpsideDown;
            return isPortrait ?
                DeviceOrientation.Portrait : DeviceOrientation.Landscape;
        }
    }
}
```

Remember the consideration about namespace naming. One of the ways to retrieve the device orientation on iOS is understanding the orientation of the system status bar, which is returned by the **`UIApplication.SharedApplication.StatusBarOrientation`** property. The logic then decides which value of the **`DeviceOrientation`** enumeration to return, based on the status. Both implementations of the partial class are returning a value from **`DeviceOrientation`**, satisfying the requirement of the base class definition.

Consuming platform-specific code

Consuming platform-specific code is extremely simple. Where needed, you declare an instance of the partial class and invoke its members as you would normally. The following is an example of the **`DeviceOrientationService`** class defined previously:

```
DeviceOrientationService deviceOrientationService =
    new DeviceOrientationService();
DeviceOrientation orientation =
    deviceOrientationService.GetOrientation();
```

At runtime, the proper implementation is automatically resolved and invoked, depending on the platform.

Differences with Xamarin.Forms

It is interesting to highlight the differences with Xamarin.Forms concerning the invocation of platform-specific code. In MAUI's predecessor, you have to define an interface with members that will be implemented by the individual platform projects.

Then, in the Android, iOS, and UWP projects, you need to write classes that implement the interface, consume native APIs, and are decorated with the **`Dependency`** attribute at the namespace level so that they can be resolved at runtime.

Finally, it is necessary to work with dependency injection to use the code by invoking the **`Get`** method of the **`DependencyService`** class, passing the interface as the type parameter, like in the following line:

```
var someData = DependencyService.Get<ICustomInterface>().DoSomething();
```

If you compare the MAUI approach with the Xamarin.Forms approach, the new one is definitely much easier and cleaner.

Wrapping native APIs: Platform integration

When accessing native APIs, most of the time, your actual need is to access features that exist cross-platform, but with APIs that are totally different from one another. For example, iOS, Android, Mac, and Windows devices all have a camera, a GPS sensor that returns the current location, and so on.

For scenarios in which you need to work with capabilities that exist cross-platform, you can leverage a set of cross-platform APIs included in the *platform integration* group. If you come from Xamarin.Forms, you might have worked with the Xamarin.Essentials library. If this is the case, platform integration in .NET MAUI can be considered the evolution of Xamarin.Essentials. However, there is no additional library, and everything you need is included in the MAUI codebase.

In this section, you will learn about some of the objects most commonly used in many applications, and you will learn about getting started with this area of .NET MAUI so that it will be easier to continue your studies with the help of the documentation.

Checking the network connection status

Tip: On Android, you need to enable the `ACCESS_NETWORK_STATE` permission in the project manifest.

One of the most common requirements in mobile apps is checking for the availability of a network connection. With .NET MAUI, this is very easy:

```
if (Connectivity.Current.NetworkAccess == NetworkAccess.Internet)
{
    // Internet is available
}
```

The **`Connectivity`** class from the **`Microsoft.Maui.Networking`** namespace provides everything you need to detect network connection and state. It is exposed as a singleton instance via the **`Current`** property. The **`NetworkAccess`** property returns the type of connection with a value from the **`NetworkAccess`** enumeration: **`Internet`**, **`ConstrainedInternet`**, **`Local`**, **`None`**, or **`Unknown`**.

The **`ConnectionProfiles`** property from the **`Connectivity`** class allows you to understand, in more detail, the type of connection, as demonstrated in the following example:

```
IEnumerable<ConnectionProfile> profiles =
                                Connectivity.Current.ConnectionProfiles;
if (profiles.Contains(ConnectionProfile.WiFi))
{
    // WiFi connection
}
```

This is very useful because it allows us to understand the following type of connections: **`WiFi`**, **`Ethernet`**, **`Cellular`**, **`Bluetooth`**, and **`Unknown`**. The **`Connectivity`** class also exposes the **`ConnectivityChanged`** event, which is raised when the status of the connection changes. You can declare a handler in the usual way:

```
Connectivity.Current.ConnectivityChanged += Connectivity_ConnectivityChanged;
```

Then you can leverage the **`ConnectivityChangedEventArgs`** object to understand what happened, and react accordingly:

```
private async void Connectivity_ConnectivityChanged(object sender,
                    ConnectivityChangedEventArgs e)
{
    if (e.NetworkAccess != NetworkAccess.Internet)
    {
        await DisplayAlert("Warning", "Limited internet connection", "OK");
        // Do additional work to limit network access...
    }
}
```

As you can see, this feature simplifies one of the most important tasks an app must do that otherwise would be done by writing specific code for each platform.

Opening URIs

It is common to include hyperlinks within the user interface of an app. .NET MAUI provides the **Browser** class, which makes it simple to open URIs inside the system browser. You use it as follows:

```
await Browser.OpenAsync("https://www.microsoft.com");
```

It is also possible to specify a preference about how to open the browser by passing a value from the **BrowserLaunchMode** enumeration as follows:

```
await Browser.OpenAsync("https://www.microsoft.com",
      BrowserLaunchMode.SystemPreferred);
```

External forces the app to launch the default browser app, whereas **SystemPreferred** opens the browser that is preferred by the system. When you use the latter, you can even customize the browser appearance by passing an instance of the **BrowserLaunchOptions** class as follows:

```
Uri uri = new Uri("https://www.microsoft.com");
BrowserLaunchOptions options = new BrowserLaunchOptions()
{
    LaunchMode = BrowserLaunchMode.SystemPreferred,
    TitleMode = BrowserTitleMode.Show,
    PreferredToolbarColor = Colors.Blue,
    PreferredControlColor = Colors.Green
};
```

TitleMode allows you to show or hide the browser title bar, whereas **PreferredToolBarColor** and **PreferredControlColor** allow you to select a color for the toolbar and control bar, respectively.

MAUI also offers the **Launcher** singleton class, which allows for opening any kind of URI with the system default app. For example, the following code opens the default email client:

```
string uri = "mailto://somebody@something.com";
var canOpen = await Launcher.Default.CanOpenAsync(uri);
if (canOpen)
    await Launcher.Default.OpenAsync("mailto://somebody@something.com");
```

It is good practice to check if the system supports opening the specified URI via the **CanOpenAsync** method; if this returns **true**, you can then invoke **OpenAsync** to open the specified URI inside the default system app.

You can use the **Launcher** class to open other apps on the device, but this is not covered here. Refer to the documentation for this particular scenario.

Sending SMS messages

Sending SMS messages is straightforward with .NET MAUI. Look at the following code:

```
public async Task SendSms(string messageText, string[] recipients)
{
    var message = new SmsMessage(messageText, recipients);
    await Sms.Default.ComposeAsync(message);
}
```

The **SmsMessage** class needs the message text and a list of recipients in the form of an array of string objects. Then, the **ComposeAsync** from the **Sms** singleton class will open the system UI for sending messages. Without this API, this would require working with platform code and with several different implementations. With MAUI, you accomplish the same result with two lines of code.

More platform integration

As you can imagine, it is not possible to provide examples for all the features wrapped by the platform integration APIs inside a book of the *Succinctly* series. The complete list of features with examples is available in the official documentation page, which is regularly updated when new releases are out.

Working with native views

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

Working with handlers

Handlers are classes that .NET MAUI uses to access and render native views, and that bind MAUI's views and layouts (discussed in Chapters 4 and 5) to their native counterparts. For example, the **Label** view discussed in Chapter 4 maps to a **LabelHandler** class that MAUI uses to render the native **UILabel**, **TextView**, and **TextBlock** views on iOS, Android, and Windows, respectively. Handlers are important because you can use them to access the full set of features of native views, overriding and customizing the behavior, look, and features of views themselves.

If you worked with Xamarin.Forms in the past, you could accomplish the same result by implementing custom renderers, which provide similar results in terms of customization but are much more complex to implement. As you will see in the next paragraphs, handlers in MAUI provide a much simpler way to access views' native features. Before starting with handlers in practice, it is important to understand how they work behind the scenes.

Architecture of handlers

Controls in .NET MAUI have an interface representation that provides abstract access to the control. For instance, a **`Button`** is backed by an **`IButton`** interface. Such interfaces are also referred to as *virtual views*, and they are mapped to native controls via handlers. Figure 63, taken from the official Microsoft documentation, shows how handlers map native controls to cross-platform controls with the support of virtual views.

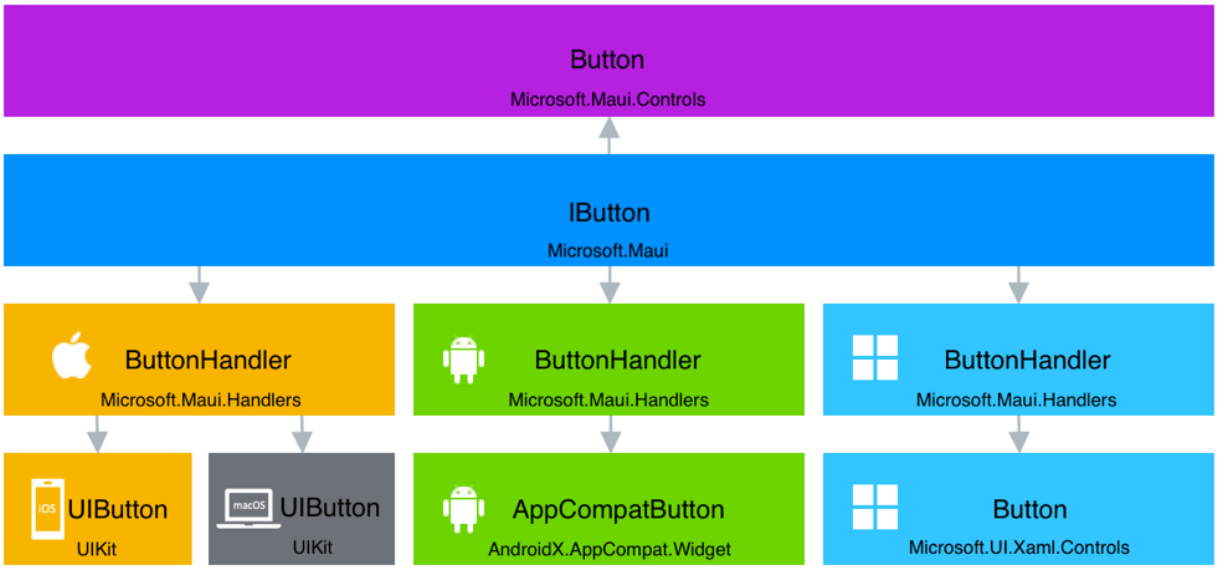

Figure 63: Handler architecture (Source: Microsoft)

In practice, you access handlers via their interface (**`IButton`** in the case of the **`Button`**). This creates a reference to the native view through a managed object. Handlers expose the full set of native features and API of a view so that you have total control over the user interface. In the next sections, you will learn how to access native features via handlers and all the aforementioned concepts will be clarified.

Tip: The documentation also shows the full list of controls based on handlers.

Customizing views via handlers

Suppose you want to implement automatic control selection on **`Entry`** views every time they receive focus. This is not possible from the shared code; the only way to do it is to leverage the native views' features. Code Listing 34 shows how to accomplish this by leveraging the **`EntryHandler`** class.

Code Listing 34

```
void ModifyEntry()
{
    Microsoft.Maui.Handlers.EntryHandler.
```

```
                Mapper.AppendToMapping("AutoSelectOnFocus",
                (handler, view) =>
        {
#if ANDROID
                handler.PlatformView.SetSelectAllOnFocus(true);
#elif iOS
            handler.PlatformView.PerformSelector(new ObjCRuntime
                .Selector("selectAll"), null, 0.0f);

#elif WINDOWS
            handler.PlatformView.SelectAll();
#endif
        });
    }
```

The following is a list of key points in the preceding code:

- The **`Microsoft.Maui.Handlers`** namespace exposes handler objects that you can use.
- In this case, the **`EntryHandler`** is used.
- The **`Mapper`** property, of type **`IPropertyMapper`**, allows for adding or changing the behavior of a feature by making it possible to access the virtual view for the current handler.
- The **`AppendToMapping`** method extends control mapping by adding a virtual property to the view via its first argument, while the second argument is an **`Action`** that allows you to invoke the API from the native view. The first argument is important because it represents a key that you can use to modify the mapping you added.
- The **`handler`** and **`view`** variables have a reference to the handler of the native view and to the cross-platform control, respectively.

Note: You can invoke the `PrependToMapping` method instead of `AppendToMapping` when you want to modify the mapping for a handler before mappings to the cross-platform control are applied. Similarly, you can invoke `ModifyMapping` to modify an existing mapping.

The **`PlatformView`** property of the handler class exactly represents the native view, and it allows for invoking its native APIs. By using preprocessor directives, you can invoke the proper native code, depending on the platform your app is running on.

On Android, autoselection is performed by invoking the **`SetSelectAllOnFocus`** method. On iOS, autoselection can be done by invoking the **`PerformSelector`** method and passing the area to be selected. On Windows, the invocation of **`SelectAll`** will autoselect the text.

Obviously, much more complex customizations are possible via handlers, but you should already have an idea of how simpler it is working with handlers in MAUI, compared to custom renderers in Xamarin.Forms. The preceding example will affect the default **`Entry`** view and all its instances in your page. You can restrict the handler's behavior to custom objects. For example, you can create a custom entry like this:

```
public class AutoSelectEntry: Entry
{

}
```

And then you can apply the autoselection to this type only, by adding an **if** block as follows:

```
    void ModifyEntry()
    {
        Microsoft.Maui.Handlers.EntryHandler.
            Mapper.AppendToMapping("AutoSelectOnFocus", (handler, view) =>
            {
                  if (view is AutoSelectEntry)
                  {
#if ANDROID
            handler.PlatformView.SetSelectAllOnFocus(true);
#elif iOS
            handler.PlatformView.PerformSelector(new ObjCRuntime
                  .Selector("selectAll"), null, 0.0f);

#elif WINDOWS
            handler.PlatformView.SelectAll();
#endif
                   }
             });
    }
```

This will still work because the **AutoSelectEntry** view relies on the **EntryHandler** and its **PlatformView** implementation.

Understanding handler events

Handlers have a lifecycle represented by the following two events:

- **HandlerChanging**, which is fired when a new handler is being created for a MAUI control and when an existing handler is being removed from a MAUI control
- **HandlerChanged**, which is fired when a MAUI control has been created. This ensures that a handler for the specified control is available.

These events are important because their handler methods can be the place to subscribe to native views' events. The following code demonstrates how to subscribe to the **FocusChange** event of the native Android text box:

```
    private void Entry_HandlerChanged(object sender, EventArgs e)
    {
#if ANDROID
        ((sender as Entry).Handler.PlatformView as
          Android.Views.View).FocusChange += OnFocusChange;
```

```
#endif
    }
```

Notice how the code only works with the platform specified via the preprocessor directives. The instance of the native view is retrieved via a double conversion: the first conversion is from the native **View** instance into a **PlatformView** object (which was explained previously), and the second conversion is converting the **PlatformView** into the actual view, an **Entry** in this case.

In the same code block, an event handler for the **FocusChange** event is specified. An example of the event handler is provided as follows:

```
private void OnFocusChange(object sender, EventArgs e)
{
    var nativeView = sender as
        AndroidX.AppCompat.Widget.AppCompatEditText;

    if (nativeView.IsFocused)
        nativeView.SetBackgroundColor(Colors.LightGray.ToPlatform());
    else
        nativeView.SetBackgroundColor(Colors.Transparent.ToPlatform());
}
```

This is just an example, and the logic inside the event handler is completely up to you, but the relevant point here is that the sender object represents an instance of the native view. In the case of an **Entry**, it is the **AppCompatEditText**.

The subscription to the **FocusChange** event has been done when the **HandlerChanged** event has been fired. This means that you can subscribe to native views' events when such an event is raised. On the other side, you need to unsubscribe from the same events when the handler is being removed, and this can be done when the **HandlerChanging** event is fired. The following is an example:

```
    private void Entry_HandlerChanging(object sender,
                HandlerChangingEventArgs e)
    {
        if (e.OldHandler != null)
        {
#if ANDROID
            (e.OldHandler.PlatformView as Android.Views.View).
             FocusChange -= OnFocusChange;
#endif
        }
    }
```

Notice how the **HandlerChangingEventArgs** class exposes a property called **OldHandler**, which is not null when the handler is being removed. In this case, you can unsubscribe from the selected events. The way you connect the **HandlerChanging** and **HandlerChanged** event handlers to the view is always the same, so in your XAML code you can assign the events with their handlers as follows:

```
<Entry HandlerChanging="Entry_HandlerChanging"
       HandlerChanged="Entry_HandlerChanged" />
```

It is quite common to interact with native views' events because this gives you a lot of control over the view lifecycle, and .NET MAUI provides you with a convenient way to do this that simplifies event handling on all the supported platforms.

Tips about partial classes

Handlers are incredibly powerful and not easy to summarize in a few paragraphs. In this chapter, you have discovered how to take advantage of handlers to customize the mapping of native views, but there is more. For example, you can write more advanced, yet cleaner code using partial classes, where you can have event handlers on one side and behavioral change on the other side.

> ***Note: If you are considering migrating a Xamarin.Forms project to MAUI, the hardest part will be migrating custom renderers to handlers. You will learn more about migration in Chapter 11, but in the meantime, keep in mind the importance of understanding handlers if you plan to move existing code to .NET MAUI.***

Introducing platform-specifics

.NET MAUI provides *platform-specifics*, which allow for consuming features that are available only on specific platforms. Platform-specifics represent a limited number of features, but they allow you to work without using handlers.

> ***Note: Platform-specifics do not represent features that are available cross-platform. They instead provide quick access to features that are available only on specific platforms. A platform-specific might be available on iOS, while the same platform-specific might not exist for Android and Windows. At the time of this writing, platform-specifics are not available for Tizen, Mac Catalyst, and tvOS.***

For instance, suppose you are working on an iOS app and you want the separator of a **`ListView`** to be full width (which is not the default). With platform-specifics, you just need the code shown in Code Listing 35.

Code Listing 35

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NativeAccess.PlatformSpecificsiOSPage"
             xmlns:iOS="clr-
namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembl
y=Microsoft.Maui.Controls"
             Title="PlatformSpecificsiOSPage">
    <VerticalStackLayout>
```

```
        <ListView iOS:ListView.SeparatorStyle="FullWidth"
             x:Name="ListView1">
            <!-- Bind your data and add a data template here... -->
        </ListView>

    </VerticalStackLayout>
</ContentPage>
```

In the case of iOS, you need to import the **Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific** namespace. Then you can use attached properties provided by this namespace on the view of your interest. In the example shown in Code Listing 37, the attached property **ListView.SeparatorStyle** allows you to customize the separator width.

Platform-specifics can also be used in C# code. In this case, you need two **using** directives to import the **Microsoft.Maui.Controls.PlatformConfiguration** and **Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific**. Then you can invoke the **On** method on the view of your interest, passing the target platform and supplying the platform-specific implementation you need.

The following code provides an example that represents the same scenario seen in Code Listing 36, but in C# code:

```
this.ListView1.On<iOS>().SetSeparatorStyle(SeparatorStyle.FullWidth);
```

Platform-specifics work the same way on Android and Windows. In the case of Android, the namespace you import is **Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific** (for both XAML and C#), whereas for Windows the namespace is **Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific**.

Code Listing 36 shows an example of Android platform-specifics that allow you to enable fast scrolling on a **ListView**.

Code Listing 36

```
<?xml version=”1.0” encoding=”utf-8” ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NativeAccess.PlatformSpecificsAndroidPage"
             xmlns:android="clr-
namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;ass
embly=Microsoft.Maui.Controls"
             Title="PlatformSpecificsAndroidPage">
    <VerticalStackLayout>
        <ListView android:ListView.IsFastScrollEnabled="true"/>

    </VerticalStackLayout>
</ContentPage>
```

The documentation provides lists of built-in platform-specifics for iOS, Android, and Windows.

Tip: A platform-specific for one platform will simply be ignored on other platforms.

Chapter summary

Applications often need to work with features that you can only access through native APIs. .NET MAUI provides access to the entire set of native APIs on iOS, Android, and Windows via several options. With the **`DeviceInfo`** class, you can get information on the current system from your shared code. By implementing an abstract class on multiple platforms, you can resolve cross-platform abstractions of platform-specific code in your shared code.

With platform integration, you have ready-to-use cross-platform abstractions for the most common scenarios, such as accessing sensors, network information, settings, and battery status.

You can also leverage handlers to change the look and feel of your views, and you can use platform-specifics to quickly implement a few features that are available only on specific platforms. Each platform also manages the app lifecycle with its own APIs.

Fortunately, .NET MAUI has a cross-platform abstraction that makes it simpler, as explained in the next chapter.

Chapter 10 Managing the App Lifecycle

The application lifecycle involves events such as startup, suspend, and resume. Every platform manages the application lifecycle differently, so implementing platform-specific code in iOS, Android, Tizen, Mac Catalyst, and Windows projects requires some effort.

Luckily, .NET MAUI allows you to manage the app lifecycle in a unified way and takes care of performing the platform-specific work on your behalf. This chapter provides a quick explanation of the app lifecycle and how you can easily manage your app's behavior.

Note: The app lifecycle in .NET MAUI is very different from Xamarin.Forms. Do not expect similarities in how you handle events. It is worth mentioning that in .NET MAUI, you have even more granularity of control over the application events.

Introducing the App class

The **`App`** class is a singleton class that inherits from **`Application`** and is defined inside the **App.xaml.cs** file. It can be thought of as an object that represents your application running and includes the necessary infrastructure to handle resources, navigation, and the application lifecycle. If you need to store some data in variables that should be available to all pages in the application, you can expose static fields and properties in the **`App`** class.

At a higher level, the **`App`** class exposes some fundamental members that you might need across the whole app lifecycle: the **`MainPage`** property you assign with the root page of your application, and methods you use to manage the application lifecycle, which are described in the next section.

Managing the app lifecycle

In .NET MAUI, the object that handles the application lifecycle is called **`Window`**. When the application starts up, an instance of this class is created. Such an instance also allows you to subscribe to events that represent moments of the lifecycle, summarized in Table 9.

Table 9: Events in the application lifecycle

Event	Description
`Created`	Raised after the native window (or page for mobile apps) has been created. The cross-platform window will have a handler to the native one, but it might not be visible yet.

Event	Description
Activated	Raised when the window (or page for mobile apps) has been activated and focused.
Deactivated	Raised when the window, or page for mobile apps, has lost focus (though it might still be visible).
Stopped	Raised when the window, or page for mobile apps, is no longer visible.
Resumed	Raised when an app resumes after being stopped. This event can only be raised if the **Stopped** event was previously raised.
Destroying	Raised when the native window (or page) is being destroyed and deallocated.

There are two ways to handle the app lifecycle events: overriding the **CreateWindow** method in the **App** class, and subclassing the **Window** class. Both methods are described next.

Overriding the CreateWindow method

The MAUI runtime invokes a method called **CreateWindow** to generate an instance of the **Window** class when the application starts up. You can override this method to subscribe to the app lifecycle events. Code Listing 37 shows an example.

Code Listing 37

```
namespace Chapter10_AppLifecycle;

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

            MainPage = new AppShell();
      }

      protected override Window
            CreateWindow(IActivationState activationState)
      {
            Window window = base.CreateWindow(activationState);
            window.Created += (sender, eventArgs) =>
```

```
            {
                  // take actions here...
            };

            window.Activated += (sender, eventArgs) =>
            {
                  // take actions here...
            };
            return window;
      }
}
```

What you need to do is subscribe to one or more of the app events. As an example, the preceding code starts listening to the **`Created`** and **`Activated`** events (see Table 9). You will need to add your own logic to every event handler you subscribe to, based on the type of event. For instance, if you were subscribing to the **`Destroying`** event, in the event handler, you could write code that stores the states of the app and its data.

Subclassing the Window class

An alternative to handling events in the **`App`** class directly is creating a class that derives from **`Window`** and overrides the methods that represent events. As normally happens with naming conventions in .NET, event names are preceded by the **`On`** prefix, so you will have **`OnCreated`**, **`OnActivated`**, **`OnStopped`**, and so on.

Code Listing 38 shows an example of subclassing the **`Window`** class.

Code Listing 38

```
namespace Chapter10_AppLifecycle
{
    public class MyWindow: Window
    {
        public MyWindow(): base()
        {

        }

        public MyWindow(Page page): base(page)
        {

        }

        protected override void OnCreated()
        {
            base.OnCreated();
```

```
        }

        protected override void OnActivated()
        {
            base.OnActivated();
        }
    }
}
```

Like the technique discussed in the previous section, the example only shows how to override two events, but you will be able to handle one or more events, depending on your requirements. Add your own logic inside each event handler.

At this point, the **App** class needs to create an instance of the **Window** object via the new **MyWindow** class. This can be accomplished by overriding the **CreateWindow** method as follows:

```
protected override Window
      CreateWindow(IActivationState activationState)
{
      MyWindow window = (MyWindow)base.CreateWindow(activationState);
      return window;
}
```

The two techniques, handling events in the **App** and subclassing the **Window** class, are equally good, so the choice is completely up to you. I personally prefer to handle events in the **App** class just because of how I am used to organizing my code.

Tips about native event handlers and custom events

The events described in this section are common to all the supported platforms and can be handled in a cross-platform approach. Behind the scenes, they map native events that you can fully handle in the **MauiProgram** class.

The following code shows how you can handle native events by invoking the **ConfigureLifecycleEvents** method, which requires a **using Microsoft.Maui.LifecycleEvents;** directive:

```
      public static MauiApp CreateMauiApp()
      {
            var builder = MauiApp.CreateBuilder();
            builder
                  .UseMauiApp<App>()
                  .ConfigureFonts(fonts =>
                  {
                        fonts.AddFont("OpenSans-Regular.ttf",
                        "OpenSansRegular");
                        fonts.AddFont("OpenSans-Semibold.ttf",
```

```
                        "OpenSansSemibold");
                  }).
            ConfigureLifecycleEvents(events =>
            {
#if ANDROID
                        events.AddAndroid(android =>
                              android.OnBackPressed((activity) =>
                                    { return false; }));
#endif
            });

            return builder.Build();
      }
```

Using preprocessor directives, you can configure events only for the desired platform. The list of native events is extremely long for Android, iOS, and Windows. Moreover, the native approach is not the target of this book, so you are encouraged to read through the documentation if you plan to have deeper control over lifecycle events. You will also find explanations about defining and registering custom events, which still happens via the **`ConfigureLifecycleEvents`** method.

Sending and receiving messages

In Xamarin.Forms, you used a static class called **`MessagingCenter`** to send messages that subscribers receive and then perform an action, based on a publisher/subscriber model. In .NET MAUI, this class is no longer supported. You use a class called **`WeakReferenceMessenger`**. This class is not part of the .NET base library but is provided by a NuGet package called CommunityToolkit.Mvvm, which is backed by Microsoft. The purpose of this library is to simplify the implementation of the Model-View-ViewModel pattern by offering objects that are common to this pattern.

The first thing you need to do is install this package in the sample .NET MAUI project. Figure 64 shows how the library appears in the NuGet Package Manager user interface.

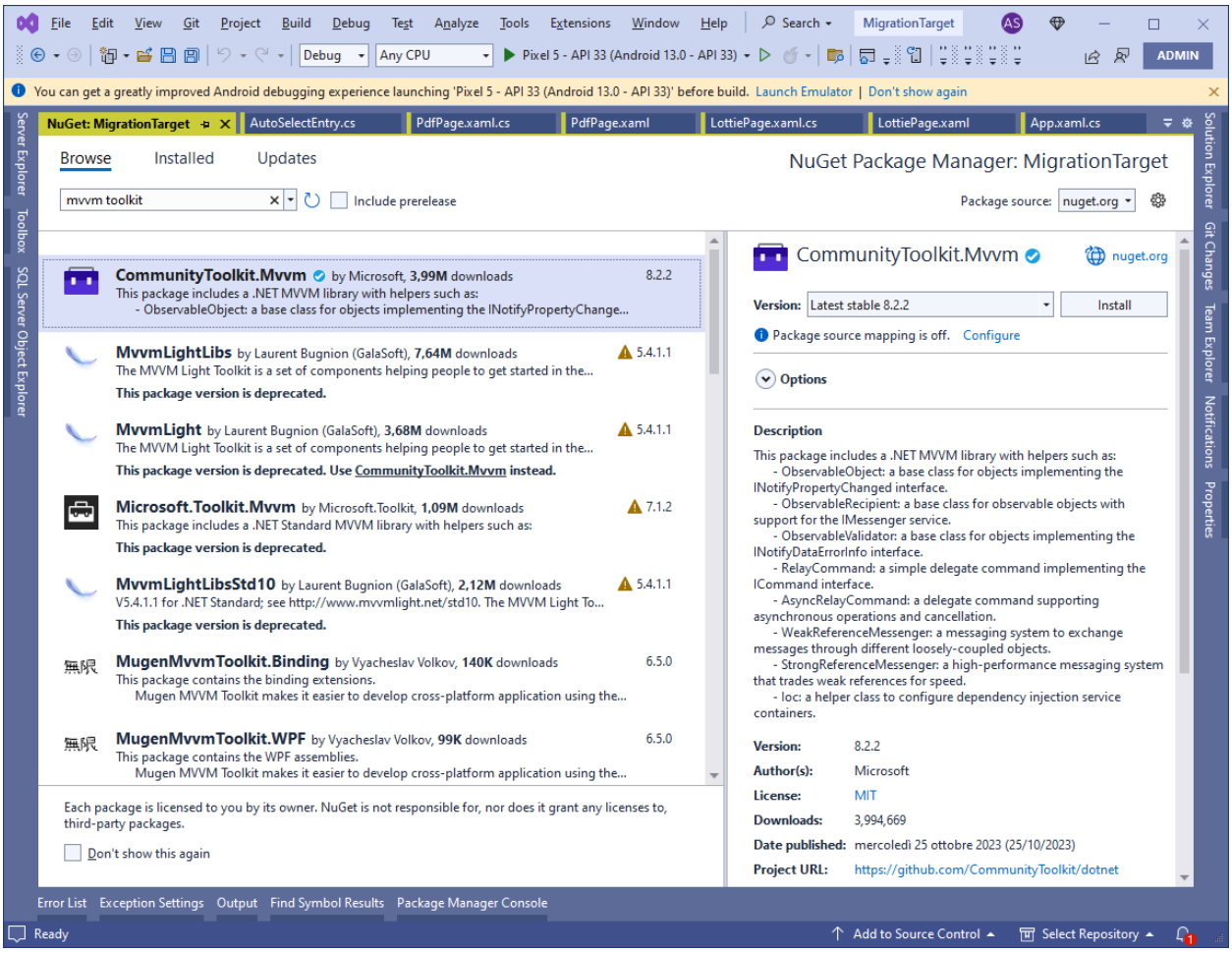

Figure 64: Installing the CommunityToolkit.Mvvm library

Once the package is installed, you are ready to start with some considerations about messages in .NET MAUI. Then, you will be able to implement your own messages.

Implementing broadcast messages

In .NET MAUI, messages are represented by a class that derives from **`ValueChangedMessage<T>`**, where **`T`** is the data type that you want to exchange with subscribers. A message is sent via the **`Send`** method of the **`WeakReferenceMessenger`** class. This is a singleton class, and its instance is exposed by the **`Default`** property. Subscribers register for messages via the **`Register`** method, which allows for taking actions when a message is intercepted. When subscribers are done, they should explicitly unregister from messages, which is accomplished via the **`Unregister`** method. Before writing code, add two new folders to the project, called **Messages** and **ViewModels**.

Defining a message

Supposing you want to send a message when a viewmodel needs to notify subscribers that an action has been completed, you first need to create a message class. So, add a new folder called **Messages** to the project. Then, add a new class file called **ActionExecutedMessage.cs** to the new folder. Code Listing 39 shows the code for the new class.

Code Listing 39

```
using CommunityToolkit.Mvvm.Messaging.Messages;

namespace Chapter10_AppLifeCycle.Messages
{
    public class ActionExecutedMessage : ValueChangedMessage<bool>
    {
        public ActionExecutedMessage(bool value) : base(value)
        {

        }
    }
}
```

In .NET MAUI, a message is not a string. Instead, it is a .NET type represented by the **`ValueChangedMessage<T>`** class, also exposed by the CommunityToolkit.Mvvm library. Every message class inherits from **`ValueChangedMessage<T>`**. You decide the data type that you want to handle. In this case, it is **`bool`**, because this is enough to understand if a button has been clicked (**`true`**) or not (**`false`**). However, you can definitely pass complex types depending on how structured the information you want to send from the message class to subscribers is. It is necessary to implement an empty constructor that invokes its base implementation.

Defining a viewmodel

The next step is implementing a viewmodel, where you will also see the message in action. Add a new folder called **ViewModels** to the project, then add a new class file called **BroadcastMessageViewModel.cs** to the new folder. Finally, add the code shown in Code Listing 40.

Code Listing 40

```
using CommunityToolkit.Mvvm.Messaging;
using Chapter10_AppLifeCycle.Messages;

namespace Chapter10_AppLifeCycle.ViewModels
{
    public class BroadcastMessageViewModel
```

```
    {
        public Command ActionCommand
        {
            get
            {
                return new Command(() =>
                {
                    WeakReferenceMessenger.
                        Default.Send(new ActionExecutedMessage(true));
                });
            }
        }
    }
}
```

This is a very simple class with only one command, which is enough for demonstration purposes. The **`ActionCommand`** property will later be bound to a button in the UI. When invoked, it sends a broadcast message. This is accomplished by invoking the **`Send`** method over a single instance of the **`WeakReferenceMessenger`** class, represented by the **`Default`** property. The method takes a message as an argument, and the constructor needs you to supply the value that you want to be sent to subscribers along with the message. Here, the code is sending **`true`**, so that callers understand that an alert will be displayed. Again, remember that you can implement and send instances of more complex types depending on the structure of your information.

Tip: While the implementation of broadcast messages is completely different in .NET MAUI, notice how the commanding implementation remains exactly the same.

Defining a new page and registering for messages

Because the purpose of the sample code is to demonstrate how to show an alert via messages, add a new content page called **BroadcastMessagesPage.xaml** to the project. At this point, add the following XAML code as the page content:

```
    <VerticalStackLayout VerticalOptions="Center"
                         HorizontalOptions="Center">
        <Button x:Name="Button1"
                Text="Click here!" WidthRequest="100"
                HeightRequest="40" Command="{Binding ActionCommand}"/>
    </VerticalStackLayout>
```

In the code-behind file, add the code shown in Code Listing 41.

Code Listing 41

```
using CommunityToolkit.Mvvm.Messaging;
using Chapter10_AppLifeCycle.Messages;
using Chapter10_AppLifeCycle.ViewModels;

namespace Chapter10_AppLifeCycle;

public partial class BroadcastMessagePage : ContentPage
{
    private BroadcastMessageViewModel ViewModel { get; set; }
    public BroadcastMessagePage()
    {
        InitializeComponent();
        ViewModel = new BroadcastMessageViewModel();
        BindingContext = ViewModel;
        WeakReferenceMessenger.Default.Register<ActionExecutedMessage>
                    (this, ButtonTapped);
    }

    private async void ButtonTapped(object recipient,
ActionExecutedMessage message)
    {
        await DisplayAlert("Info", "You clicked the button!", "OK");
    }

    protected override void OnDisappearing()
    {
        WeakReferenceMessenger.Default.Unregister<ActionExecutedMessage>
                    (this);
        base.OnDisappearing();
    }
}
```

In the constructor of the page, a new instance of the **BroadcastMessageViewModel** class is created and assigned to the page as the data source (**BindingContext**). The page also subscribes to messages of type **ActionExecuteMessage**, without knowing who the sender is. Registering to a message is accomplished via the **Register** method of the **WeakReferenceMessenger** class, which takes the message type as the generic type parameter, plus the instance of the subscriber (**this**) and a method that contains the action to take when the message is intercepted. In this case, the action is handled by the **ButtonTapped** method, which is responsible for displaying an alert. In the current example, the alert is displayed regardless of the message value, but in real-world scenarios, you might want to decide what to do based on the actual value. Finally, notice how the **Unregister** method of the **WeakReferenceMessenger** class allows for unsubscribing from message notifications. The only method argument is the instance of the subscriber object, **this** in this case. This is an important step required to avoid memory leaks.

Assigning the page to the Shell

After implementing all the necessary code, you can assign the new page to the Shell so that it is available in the navigation bar. As with the other topics, this is not mandatory in your projects, but it is good for the demonstration purposes of this book. In the **AppShell.xaml** file, replace the code that displays a page with the following:

```
<ShellContent
    Title="Message"
    ContentTemplate="{DataTemplate local:BroadcastMessagePage}"
    Route="BroadcastMessagePage" />
```

Now you are ready to run the sample project.

Running the code

If you run the sample project, you will see how the alert appears when you click the button (see Figure 65). This means that the button invoked the action from the data-bound command, which sent a broadcast message that was received by the user interface.

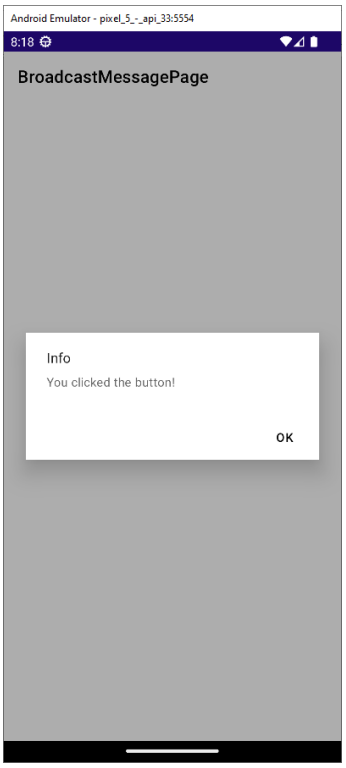

Figure 65: The message was received and the UI is reacting

As you have seen, broadcast messages in .NET MAUI are very different from Xamarin.Forms, and require a more complex implementation, but the benefit is that you can pass complex types as parameters.

Chapter summary

Managing the application lifecycle can be very important, especially when you need to get and store data at application startup or suspension. .NET MAUI prevents the need to write platform-specific code, offering a cross-platform solution through handling events inside the **`App`** class or by subclassing the **`Window`** class. Not only is this a powerful feature, but it really simplifies your work as a developer.

Finally, you have seen a new way to implement broadcast messages via the **`WeakReferenceMessenger`** class, which is useful with logic decoupled from the user interface.

Chapter 11 Migrating from Xamarin.Forms

Without a doubt, .NET MAUI is the successor of Xamarin.Forms, and most developers approaching .NET MAUI come from the Xamarin.Forms development experience. Microsoft stopped supporting Xamarin.Forms on May 1, 2024, so there are several reasons to migrate existing Xamarin.Forms solutions to MAUI: technology improvements, easier maintenance, and the end of support for Xamarin.Forms.

Microsoft provides a very detailed page about migration, with a strictly technical focus. This chapter provides, instead, information on how migrating a project works from a more general perspective.

.NET MAUI has been built in a way that migrating Xamarin.Forms solutions should be easy, but there are some caveats that you will discover in the sections that follow.

> ***Note: The end of the official support for Xamarin.Forms means that Microsoft will no longer release updates and hotfixes. This includes support for new API levels that Apple and Google often require us to target in order to keep applications available in their stores. In addition, Microsoft is no longer obliged to reply to customers' technical support requests.***

General considerations

There are a few general, common considerations that apply to each solution you want to migrate:

- Migrating Xamarin.Forms projects means migrating to .NET. Your existing code will likely work, but there might be breaking API changes that you will need to fix.
- You will need to change XAML namespace references.
- You will need to change C# namespace references.

Table 10 summarizes the namespaces you will need to update.

Table 10: Namespaces to be updated

Xamarin.Forms namespaces	.NET MAUI namespaces
`xmlns="http://xamarin.com/schemas/2014/forms"`	`xmlns="http://schemas.microsoft.com/dotnet/2021/maui"`
`Xamarin.Forms`	`Microsoft.Maui` and `Microsoft.Maui.Controls`

Xamarin.Forms namespaces	.NET MAUI namespaces
`Xamarin.Forms.Xaml`	`Microsoft.Maui.Controls.Xaml`
`Xamarin.Essentials`	`Microsoft.Maui.Essentials`

This is the easiest part. Now it is time to analyze solutions for migration.

Analyzing existing solutions

Migrating a Xamarin.Forms solution to .NET MAUI certainly has costs, especially in terms of time and human resources. So, you need to carefully analyze your existing code and accurately plan the migration. The following list describes points that you should consider before you start:

- **Third-party components**: Before you start migrating, make sure there is a .NET MAUI version of each third-party component you use in your Xamarin.Forms solution.
- **NuGet packages**: Similarly, ensure there is a .NET MAUI version of each NuGet package, and more generally, of each dependency you have in your Xamarin.Forms solutions.
- **Custom renderers**: If your Xamarin.Forms solutions include custom renderers, ensure you have an option to convert them to MAUI handlers. This might not always be possible in a simple way, and you might need to completely rearchitect your code.
- **Assets and resources**: Ensure that all your assets and resource files are still supported in .NET MAUI.

This list is not exhaustive because a lot depends on the particulars of each solution, but it is a very good starting point. If you can satisfy all the listed points, you can start considering a migration from Xamarin.Forms to MAUI.

The next paragraphs describe the two possible options available at the time of writing. If you cannot satisfy the preceding points, do not worry, and do not be in a hurry. Third-party vendors will certainly upgrade their product suites to target .NET MAUI, so it might just be a matter of waiting. After all, waiting is not so bad. As time goes on, MAUI will undergo many improvements and will be more reliable.

Note: Before you start any migration process, it is a good idea to place your existing solutions under source control if you haven't already. This will help you revert your changes easily.

The Upgrade Assistant

Microsoft is working on a tool called Upgrade Assistant, which is currently in preview, and whose goal is to help automate the migration process. The Upgrade Assistant has the more

general purpose of helping migrate existing .NET solutions to .NET 8, so it is not limited to a Xamarin.Forms-to-MAUI migration.

The Upgrade Assistant is a command-line tool that creates a backup copy of your projects and attempts to change the namespaces as described in Table 10. Because it is in a preview state, it is neither a reliable tool nor an effective solution for migration—these are the reasons why this tool is only mentioned and not covered. In the future, it should be able to upgrade NuGet package references and take further steps.

Even if it is eventually built in a step-by-step wizard way, for now you need to run the tool against every project of a Xamarin.Forms solution, and this is not convenient. Keep an eye on the development status of the tool, but if you want to start migrating now, the manual approach is the best one.

Effective migration: Creating a new project

With the resources you have today, the most effective way to migrate a Xamarin.Forms solution to .NET MAUI is a manual migration. Assuming you have analyzed the availability of .NET MAUI resources for your existing code, this is how manual migration works:

1. Create a new .NET MAUI project.
2. Set up the platform properties to match your existing Xamarin-platform project's settings.
3. Install the .NET MAUI version of all the third-party libraries, components, and NuGet packages that you also used in your Xamarin.Forms solution.
4. Copy all the asset and resource files into the .NET MAUI resource folders described back in Chapter 1.
5. Manually re-create the folder structure in your new project.
6. Manually copy all your C# code files.
7. Manually create all the necessary XAML files and copy both the XAML and C# code from the existing solution to the new one, paying attention to namespace references, as described in Table 10.
8. Fix all the API changes and code references that are different in .NET MAUI.

As you can see, migration is a very complex task, and it requires a lot of time and resources. However, if you plan to maintain your projects over a long time, it is something that you cannot avoid considering.

Chapter summary

This chapter provided practical suggestions on how to plan a migration from Xamarin.Forms to .NET MAUI, explaining all the necessary parts you must consider, such as libraries, custom renderers, and the project structure. As MAUI undergoes new releases, we can expect updates to the Upgrade Assistant tool that will help automate the process.

This book has given you the foundations for building cross-platform projects with .NET MAUI, explaining how you create and debug solutions, how you build the user interface with XAML, how you handle resources and navigation, and how you write platform-specific code, which includes managing the application lifecycle.

There is certainly much more to .NET MAUI, and updates will be released from Microsoft in the future, so it is recommended that you always keep an eye on the official documentation to stay up to date. Also, an upcoming ebook in the *Succinctly* series, dedicated to migration, will be available soon.