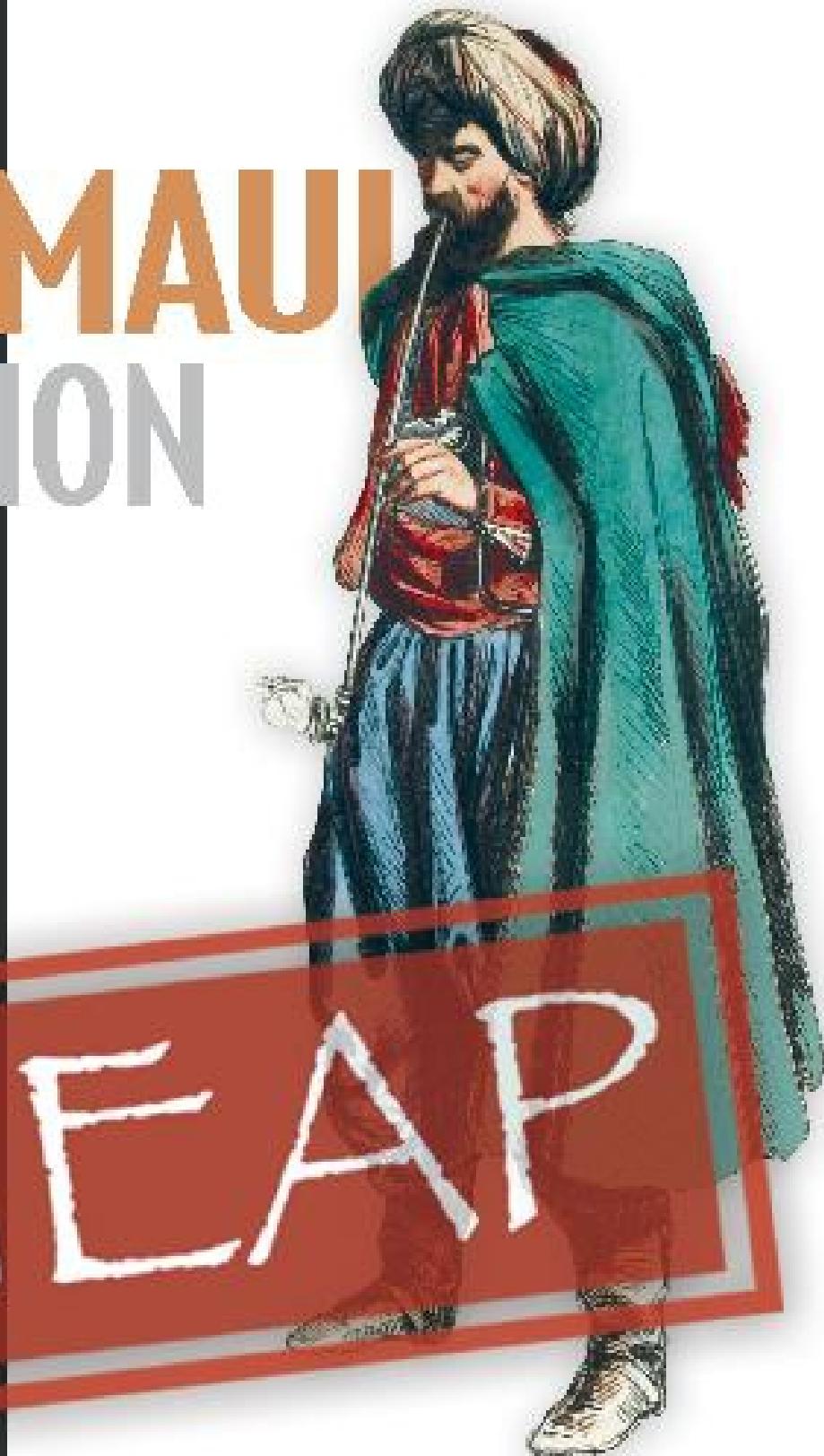


.NET MAUI IN ACTION

Matt Goldman

MEAP



MANNING

.NET MAUI IN ACTION

Matt Goldman



MANNING

.NET Maui in Action MEAP V08

1. [Copyright 2023 Manning Publications](#)
2. [welcome](#)
3. [1 Introducing .NET MAUI](#)
4. [2 Build a .NET MAUI app](#)
5. [3 Making .NET MAUI apps interactive](#)
6. [4 Controls](#)
7. [5 Layouts](#)
8. [6 Advanced layout concepts](#)
9. [7 Pages and navigation](#)
10. [8 Enterprise app development](#)
11. [9 The MVVM pattern](#)
12. [10 Styles, themes, and multi-platform layouts](#)
13. [11 Beyond the Basics: Custom Controls](#)
14. [12 Deploying Apps to Production with GitHub Actions](#)
15. [Appendix. A Setting up your environment for .NET MAUI development](#)
16. [Appendix B. Upgrading a Xamarin.Forms app to .NET MAUI](#)



MEAP Edition

Manning Early Access Program

.NET MAUI in Action

Version 8

Copyright 2023 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes.

These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/dot-net-maui-in-action/discussion>

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for purchasing the MEAP edition of *.NET MAUI in Action*.

I came to .NET MAUI via Xamarin, and when I first started learning Xamarin, it was because I wanted to build apps, not because I wanted to become a Xamarin expert. I quickly found, though, that a level of expertise was required to achieve even the simplest tasks; and not just expertise in Xamarin, but expertise in the iOS and Android platform APIs too.

Xamarin, and especially Xamarin.Forms, has come a long way since then, and building rich, functional cross-platform apps with Xamarin.Forms as very achievable for .NET developers. But .NET MAUI takes this even further by providing a top down, .NET first API for .NET developers to build mobile and desktop apps, by leveraging their existing skills to get started right away.

Building expertise in a topic is valuable, but I don't believe it should be a prerequisite to getting started. My goal with this book is to teach you the fundamentals you need to get started with .NET MAUI right away. As you progress through the book we'll add more skills to your toolchest, and by the end of the book you'll know how to build some cool mobile and desktop apps, but you'll also know exactly where to go to learn more, as and when you need to.

I love building software, but there's something extra special about mobile apps. Being able to physically hold your creation in your hand and interact with it through touch gives me an elevated sense of satisfaction. Building web and cloud products is cool too, but mobile, and to nearly the same degree desktop, development is on another level for me.

I hope that this book helps me to share some of that enthusiasm with you, and I hope you find an appreciation for building mobile and desktop apps by building on a foundation of your existing skills in .NET.

As .NET MAUI is still somewhat in its infancy, as you progress through the

book, you may encounter some issues with the code that are not just errors on my part (although feedback for my own errors is of course the most valuable feedback you could give me).

At time of writing, there are over 1,600 open issues on the .NET MAUI repository on GitHub. Most of these are trivial and none of them are showstoppers, but if you encounter a problem in the book it's worth checking here first: <https://github.com/dotnet/maui/issues>. If you encounter an issue and find it's been raised there, your upvote will help to push it up the queue. And if you encounter a new issue that's not been logged yet, logging it on GitHub would be doing an immense service to the .NET MAUI community. Alternatively, I am happy to log issues on your behalf.

It's important to me to make .NET MAUI approachable and accessible to .NET developers, so your feedback via [liveBook Discussion Forum](#) will be invaluable as I write this book.

Thank you for being a part of this journey!

Matt Goldman

In this book

[Copyright 2023 Manning Publications](#) [welcome](#) [brief](#) [contents](#) [1 Introducing .NET MAUI](#) [2 Build a .NET MAUI app](#) [3 Making .NET MAUI apps interactive](#) [4 Controls](#) [5 Layouts](#) [6 Advanced layout concepts](#) [7 Pages and navigation](#) [8 Enterprise app development](#) [9 The MVVM pattern](#) [10 Styles, themes, and multi-platform layouts](#) [11 Beyond the Basics: Custom Controls](#) [12 Deploying Apps to Production with GitHub Actions](#)
[Appendix. A Setting up your environment for .NET MAUI development](#) [Appendix B. Upgrading a Xamarin.Forms app to .NET MAUI](#)

1 Introducing .NET MAUI

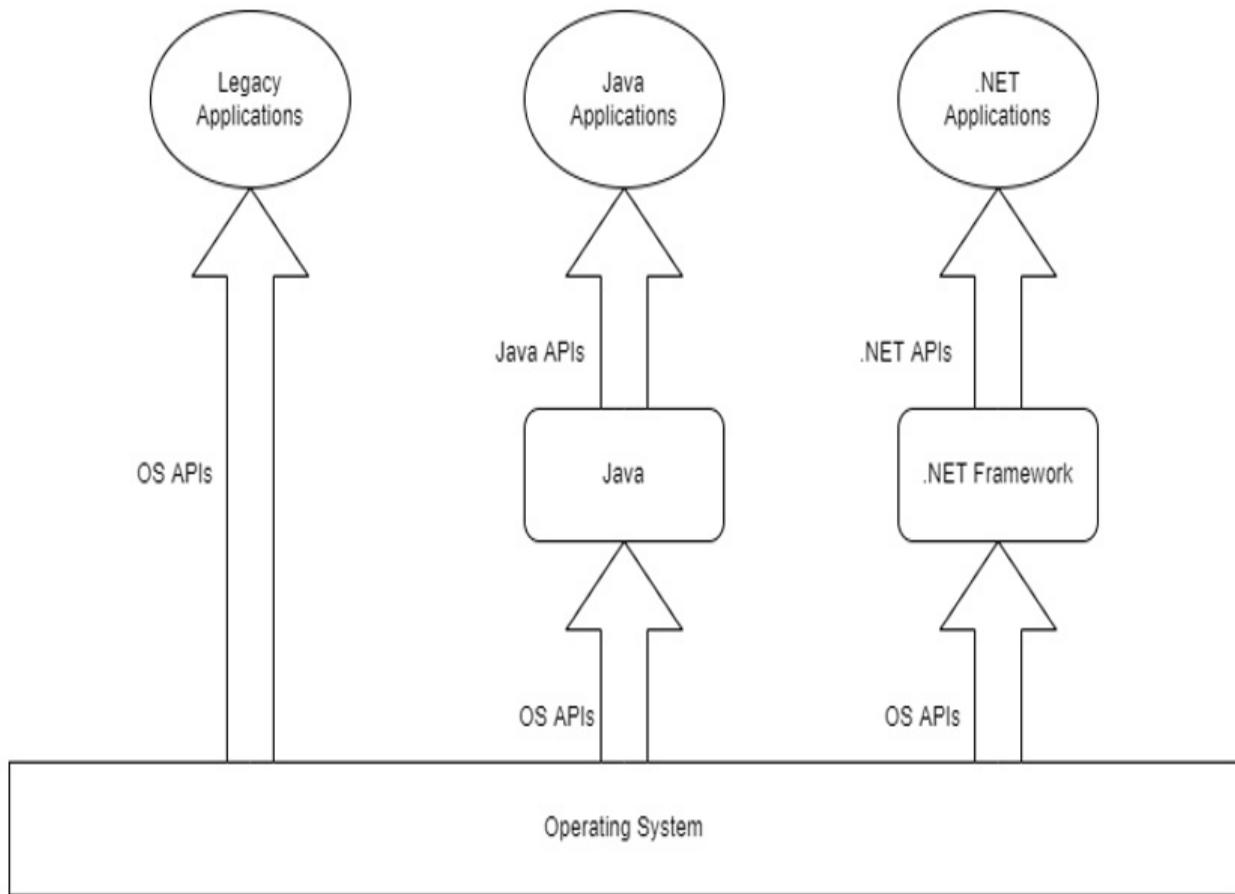
This chapter covers:

- What .NET MAUI is
- How MAUI fits into .NET
- Why you would want to use .NET MAUI to build desktop and mobile apps
- Writing cross-platform UIs

The dream of write once, run anywhere (WORA) cross-platform software began in earnest in 1996 with the release of the first version of Java by Sun Microsystems. Before Java, software developers could only write their code against APIs provided by the operating system. Java was different – not only was it a new programming language, but it was also a runtime with its own set of APIs, allowing developers to ignore the target platform or operating system. Sun provided a runtime for nearly every available operating system (called the Java Virtual Machine, or JVM), which meant that developers didn't have to worry about whether they were building a Windows application, a Unix application, a Linux application, or a Mac application. They were building a Java application.

Microsoft started its own journey toward a cross-platform runtime shortly afterwards, releasing the first public version of the .NET Framework in 2000. It wasn't cross-platform, but the development paradigm was similar – developers didn't have to write code against Windows APIs anymore, they used the .NET Base Class Library (BCL) to write code using the .NET APIs. Like the JVM, the .NET Framework was a runtime that was installed independently of the operating system, so developers didn't have to worry about what version of Windows their users had installed, they just needed to ensure they had the right version of the .NET Framework.

Figure 1.1 Legacy applications are built directly on top of operating system APIs. Java and .NET provide their own APIs for developers to use and provide runtimes that hide the platform APIs.



But there was still one problem with this approach – the .NET Framework was Windows only. Developers writing .NET applications couldn't target the Mac or Linux platforms, both of which were gaining momentum. This didn't really change, at least from Microsoft's perspective, until 2016 with the release of .NET Core. .NET Core diverged from .NET Framework in a few ways, but critically by stripping out key Windows dependencies, and abstractions for Windows APIs. Unlike .NET Framework, which had to be installed and would only work on Windows, .NET Core is a truly portable runtime that can be shipped alongside the code that it executes, and runs on Windows, Mac and Linux.

While this was a huge step closer to the WORA dream, UI applications were still missing from the picture. .NET Core applications are command line only (this includes web servers and other services). .NET Framework provided platforms for developing UI applications for Windows, initially with Windows forms and later the Windows Presentation Foundation (WPF) but,

being Windows specific, these were never brought across to .NET Core.

Outside of Microsoft, the journey towards cross-platform UI applications in .NET took on a life of its own. Within a year of release of the first version of the .NET Framework, the .NET specification became an open standard. Open standards drive the modern web and enable the development of competing or complimentary runtimes. For example, because HTML, JavaScript and CSS are open standards, anyone can build a web browser, and developers and users have a choice of which technologies to use. While the journey to .NET being fully open was by no means a straight line nor without its bumps, opening the .NET standard enabled Miguel de Icaza, working at Novell at the time, to release an open-source .NET compiler for Linux, called *Mono*.

By the end of the 2000s the iOS and Android operating systems, and more importantly their application distribution platforms (the iOS App Store and Google Play store respectively), had become well established, and any discussion around cross-platform UI applications became dominated by mobile. iOS and Android use different languages and paradigms for app development, and while it's possible to learn both, most developers prefer not to write and maintain multiple versions of their software if they don't have to.

Mono was ported to iOS in MonoTouch and to Android in MonoDroid. These eventually evolved into Xamarin, which provided not only a .NET compiler for iOS and Android, but a complete abstraction of the iOS and Android APIs in .NET. Using Xamarin, you still had to learn the iOS and Android APIs, but then you would write your code in a .NET language like C# instead of in a vendor provided language (like Objective-C or Swift for iOS or Kotlin for Android). Using Xamarin, you could also share all your non-UI code between the two platforms, so any business logic or code for communicating with a back end could be written once and used in a Xamarin.iOS project and a Xamarin.Android project.

In 2014 Xamarin introduced Xamarin.Forms, which provided an API for writing cross-platform UI code using extensible application markup language (XAML), the markup language originally introduced in WPF. This allowed developers to share not just business logic, but UI as well, across iOS, Android and UWP (and, at the time, Windows Phone). Just like the other abstractions, the UI code you write in XAML is an abstraction, and when you

compile your app for iOS, your XAML code gets interpreted into iOS' native UI, and when you compile for Android, the XAML is compiled to native Android UI code.

Xamarin was acquired by Microsoft in 2016 and has been under active development there ever since. Xamarin.Forms is now a stable and mature product, used to build many successful enterprise and consumer applications. But the current version, Xamarin.Forms 5, will be the last. Coming in its place is .NET MAUI, which is described by Microsoft as the next evolution of Xamarin.Forms. While .NET MAUI shares a lot of its DNA with Xamarin and Xamarin.Forms, it is an entirely new platform built from the ground up to usher in a new era of truly cross-platform, WORA applications written in .NET.

It's an exciting time to be a .NET developer.

1.1 What is .NET MAUI?

The .NET Multiplatform App UI (MAUI) is a new framework from Microsoft for building cross-platform UI applications that target Windows, macOS, iOS and Android.

With .NET MAUI, you can build a rich, interactive, native UI application that runs on any one of these platforms. With a single code base, you can build an application that supports all these platforms and share 100% of the code between them. In short, you write an application in a .NET language, and it runs without any changes on any of the target platforms. All your logic can be written in a .NET language, and your UI can be defined in either XAML or in your .NET language of choice too.

.NET MAUI Development Languages

You can write code in .NET MAUI apps using your .NET language of choice (VB.Net, F# or C#). It's possible to define your UI in code too, but only C# and F# are supported.

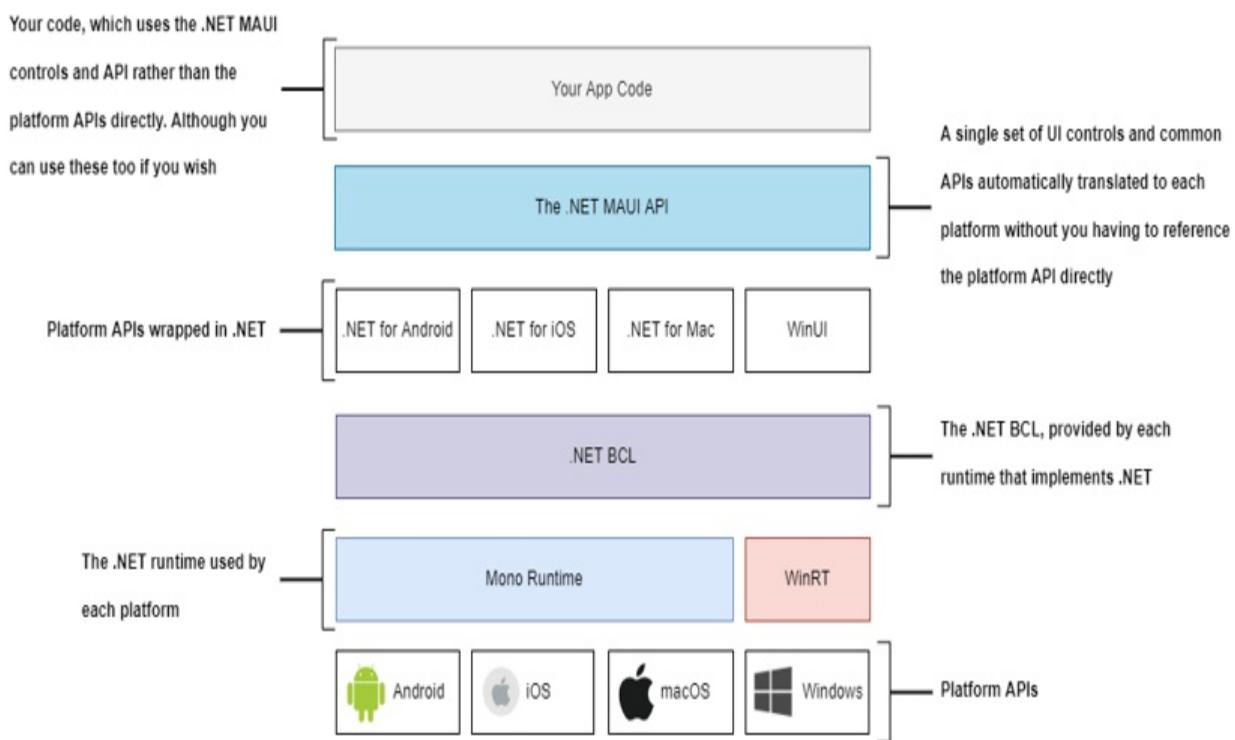
In this book, we will be using C# to write our logic and XAML for defining

our UI (although there will be some exceptions) as these are the most common approaches.

It should be possible to adapt the C# code to another language, but I would recommend working through the examples in the book in C# first.

Figure 1.2 shows the architecture of a .NET MAUI application, and we will examine the layers and steps to see how a .NET MAUI application comes together.

Figure 1.2 .NET MAUI is built bottom-up; each platform provides APIs, and there is a .NET runtime for each platform (WinRT on Windows, Mono on everything else) that is built on these APIs. Each layer provides APIs used to build the APIs in the layer above. Meanwhile, your code is written top-down; you write a .NET MAUI app, and the architecture takes care of encapsulating it for lower layers.



Let's start by looking at the layers in this diagram. At the bottom we have the target operating system (Android, iOS, macOS and Windows). The next layer up shows the .NET runtime that will execute our .NET MAUI app on each target OS. For Android, iOS and macOS, this runtime is Mono, and for Windows it is WinRT.

The next layer up is our first abstraction – the .NET Base Class Library (BCL). The BCL provides access to all the common language features we would expect, such as lists and generics, that don't form part of .NET's primitives. From .NET 5 onwards, .NET (without Core or Framework) has become the new standard, replacing the .NET Standard too. From a developer perspective, .NET 5 and .NET 6 have become target frameworks, replacing `netcoreapp` and `netstandard`. When writing a .NET MAUI app in .NET 6, you have access to the BCL across all platforms.

The next layer, which sits on top of the BCL, provides access to abstractions for platform specific APIs. .NET for Android and .NET for iOS are the next iterations of Xamarin.Android and Xamarin.iOS respectively. These are bindings to the platform APIs, using the same types and namespaces used by Objective-C, Swift, Java or Kotlin developers. .NET for Mac is new but operates the same way, and for Windows, the WinUI API is used. This includes everything available in each platform's API, from simple layouts and controls like buttons and text entry fields to more sophisticated APIs like ARKit on iOS and ARCore on Android, for developing augmented reality (AR) applications..

The last abstraction is .NET MAUI. This is a unified API that provides UI elements that are common to all supported platforms. This includes various views like layouts, buttons, text entry fields, navigation APIs, and many more. Through the `Essentials` namespace (in Xamarin this was started as a separate package), you also get access to common hardware features, such as Bluetooth, location services, and device storage.

While we looked at the layers bottom-up, the philosophy of building a .NET MAUI app is very much considered top-down:

1. You build a cross-platform application by writing .NET MAUI code (rather than, say iOS or Android code).
2. If you want to, you can still write platform or OS specific code in your application, but you don't have to.
3. .NET MAUI takes your code and compiles it for the target platform. Understanding how .NET MAUI builds your application for various platforms is not necessary to build a .NET MAUI application, although having a good understanding of these platforms is beneficial. Not only

will you be better able to troubleshoot OS or platform specific errors, but you'll also open up the entire spectrum of platform APIs, not just those exposed in top-level .NET MAUI wrappers. We'll cover some platform specific aspects of .NET MAUI development as we go.

.NET MAUI is much more than just the next version of Xamarin.Forms. Whereas Xamarin was a software development kit (SDK) that you installed independently of .NET, MAUI is a *workload*, meaning it is a part of .NET, just like ASP or console app development. This approach provides a few benefits which we will cover later, but most importantly it demonstrates Microsoft's commitment to the future of .NET MAUI and its inclusion as a core part of .NET.

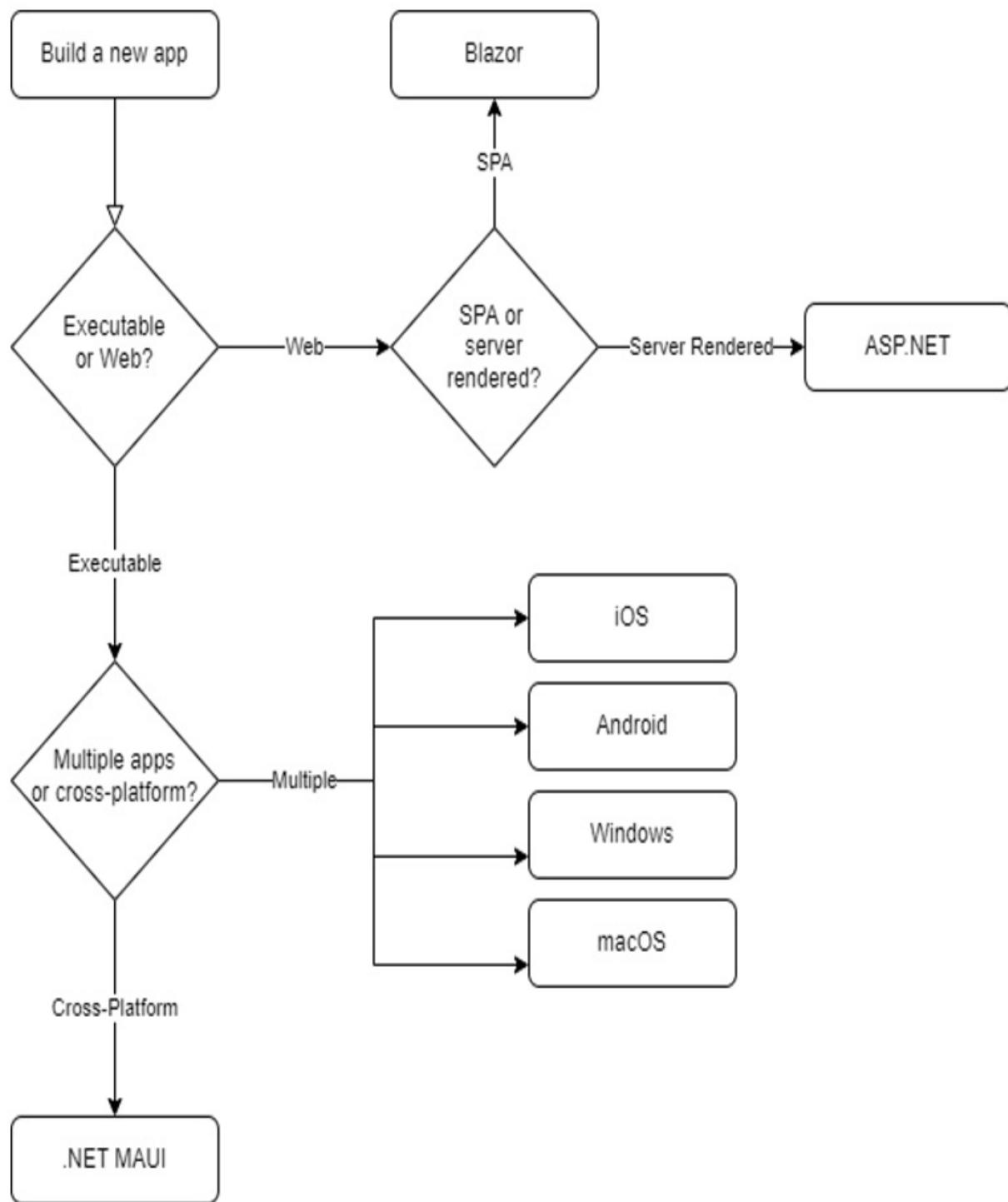
1.2 Cross-platform vs ‘native’ apps

When you decide to build an application, either as an independent developer or as part of an enterprise development team, you must ask a few questions and make some decisions.

- Will you build an installable, native binary executable application? Or will you build a web app?
- If you build a web app, will you use a single page application (SPA) framework (such as Angular or Blazor) or will you use a traditional server generated page framework (like ASP.NET or PHP)?
- If you build an installable app, will you build one for each platform where you want your users to run it, or will you build a single app that runs on every platform?

These are just some of the questions and decisions you need to resolve. Your decision process might look something like this.

Figure 1.3 Assuming you are a .NET developer, the biggest decision to make is whether you want to build a web app or an installable/executable app. If you choose to build an executable, .NET MAUI is a no-brainer.



To help with this decision, it's important to understand something (and potentially dispel a myth): apps built with .NET MAUI *are* native apps. That is to say, any app that you write with .NET MAUI compiles to a native binary executable for each target platform. The same on iOS as if it were written in

Swift, the same on Android as if it were written in Kotlin (although .NET code is still JIT compiled by default, you can enable AOT).

With that in mind, perhaps a bigger decision is not whether to use a cross-platform framework such as .NET MAUI or use vendor provided languages, but rather whether to build a binary application at all as opposed to a web app (although with .NET MAUI Blazor you get the best of both worlds, as we'll see in section 1.4). That decision will be up to you and/or your team and will depend on many factors. Both approaches have pros and cons; and while web app development certainly has a lot of advantages for many situations, there are still some compelling reasons to choose a binary application instead.

These include:

- **Multi-threading.** Applications running in a web browser can only use one thread at a time. Depending on your performance requirements, this may not be an issue.
- **Encryption.** Web applications use encryption to communicate with back-end services, but you can't securely store data offline in a web browser.
- **Access to device hardware features.** Many hardware features are available to browsers now, such as camera and location services, and even Bluetooth. Other features, such as telephony or (SMS) messaging, are either difficult or impossible to access from a browser app. But providing access to these consistently and reliably is much easier using an installed binary application.
- **Access to platform APIs.** You might want to access certain platform features, such as ARKit on iOS or ARCore on Android which would influence your decision to choose an installed binary application rather than a web app.

This is by no means an exhaustive list. There are many factors which may influence your decision to choose one approach rather than the other.

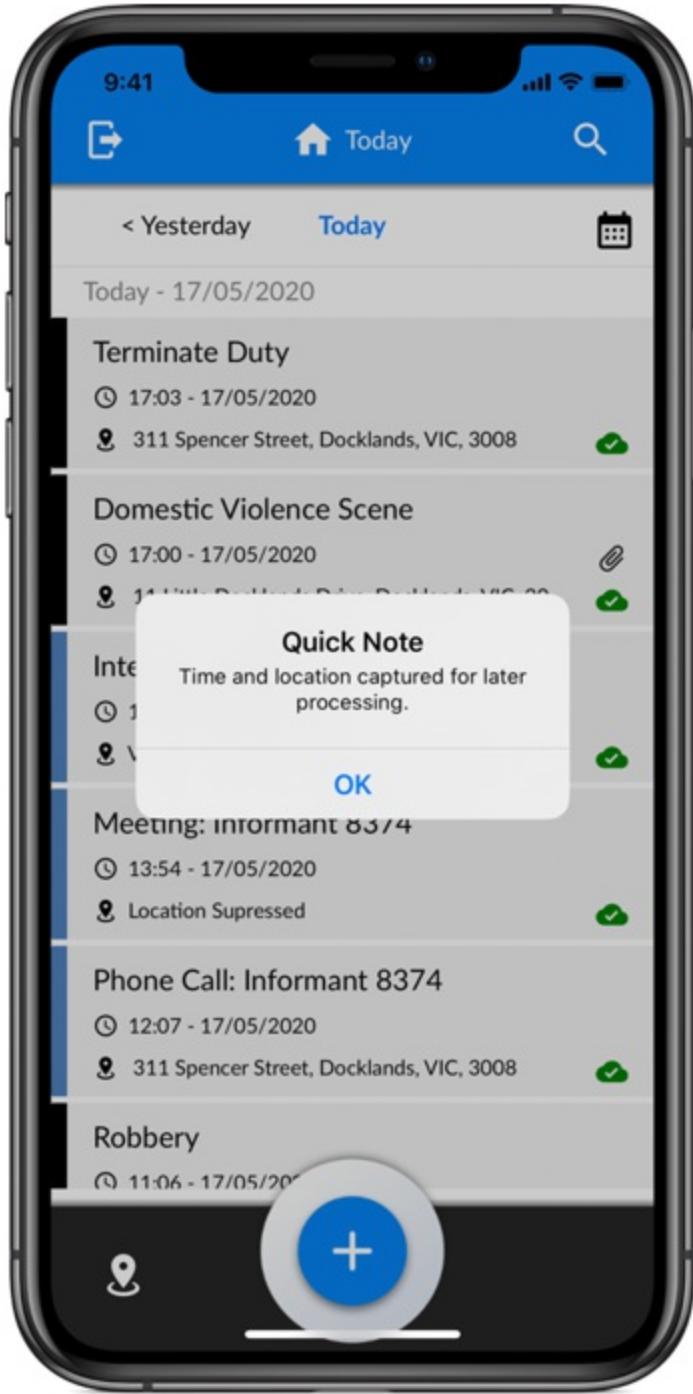
However, all other things being equal, perhaps the most compelling reason to choose to build an installable app rather than a web app is branding. Having an app store presence is considered critical for most businesses that wish to reach a broad sector of the market, and store presence provides a level of trust to your users that, rightly or wrongly, may not be there with a web app alone.

Desire to have a presence in the App Store and Google Play store is often a motivating factor revealed by my clients when I ask them why they want a mobile app.

Nevertheless, the reasons mentioned above can be compelling. Multi-threading can be important for performance intensive applications. Browser-based apps can simulate multi-threading, but in a binary app running on hardware with a multi-core CPU, those threads can actually run simultaneously, meaning background processes don't lock up the UI.

Being able to encrypt data at rest as well as in motion can be an important consideration if security is a chief concern or feature of your app, and this requirement would rule out building a browser-based app altogether. Additional platform features, such as biometric identification, also add an extra layer of security (as well as convenience) to binary apps.

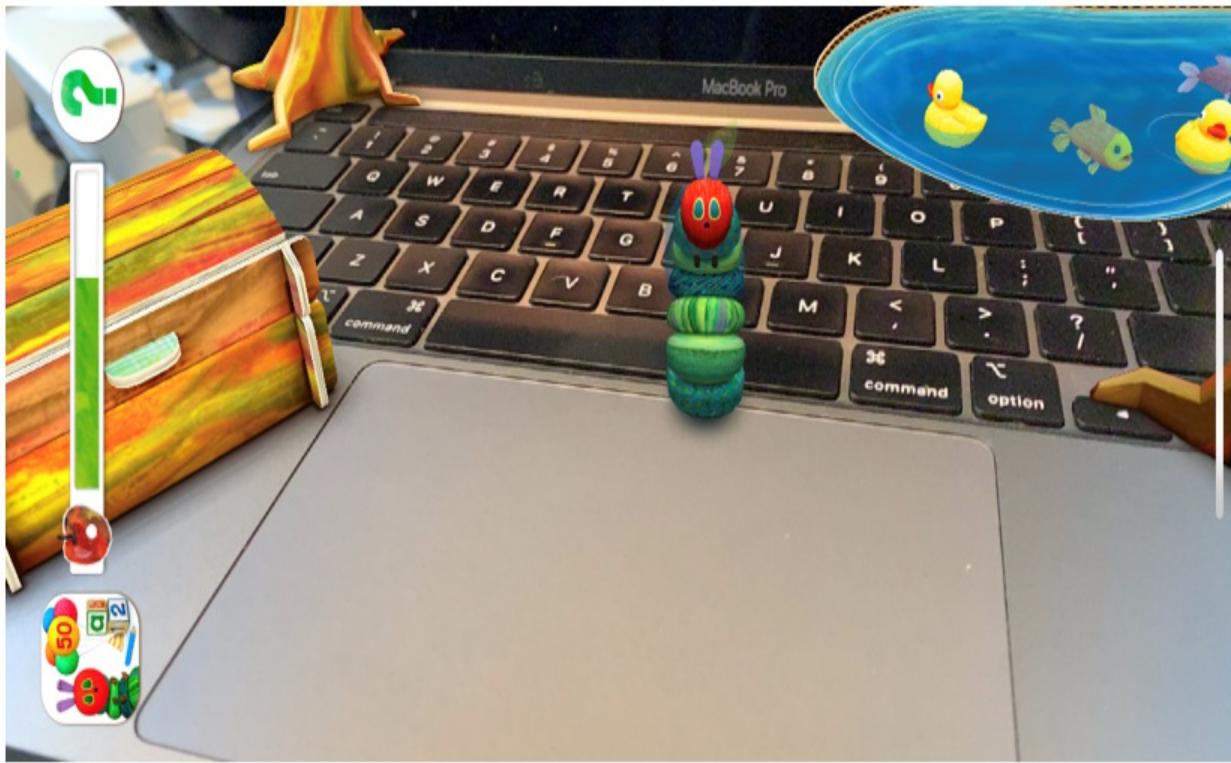
Figure 1.4 The Verinote mobile app was built for security from the ground up. Data captured on the device is encrypted until it can be synced with a cloud service. This would not have been achievable with a web app.



Verinote, an app built in Xamarin.Forms and currently being upgraded to .NET MAUI, takes advantage of these security features. Designed for law enforcement (or any regulated industry), Verinote lets users in the field capture notes with photographs, audio recordings, and sketches, and syncs the notes back to a cloud service. It needs to work offline as well as online, so if

no connection is available data is cached locally until the cloud service can be reached. Due to the highly sensitive nature of the information users of this app work with, security is a paramount concern. Verinote encrypts cached data (as well as all data in motion), and uses platform-provided biometric authentication to secure access to the app. Because of these dependencies on security features provided by native platform APIs, Verinote could not have been built as a web app (Verinote does of course have a web app component too, but it does not, and can't, store any data offline).

Figure 1.5 The Very Hungry Caterpillar enjoying a snack on my keyboard. Making augmented reality apps like this for mobile devices is simplified through the use of APIs provided by Apple and Google. Building a similar experience in a web browser would be extremely difficult, if not impossible.



Device features and platform APIs unlock almost limitless possibilities for mobile and desktop developers. I mentioned ARKit and ARCore earlier which are the iOS and Android augmented reality (AR) libraries respectively, which you can use to build rich compelling experiences and products for users. Some examples include the proliferation of AR measuring apps available, as well as awesome new retail experiences like Ikea's mobile app.

My toddler is also particularly fond of the MyCaterpillar app, which puts the caterpillar from Eric Carle's *Very Hungry Caterpillar* into your space and lets you feed and interact with it. These kinds of apps are a little beyond the scope of this book, but are nevertheless very achievable with .NET MAUI.

If you've made the decision to build a binary, installable application, the next decision is whether to build multiple versions of the app – one for each platform – or to build your product using a single cross-platform code base. This decision may seem like a no-brainer, and for most cases it probably is. There may be some niche scenarios where your requirements are specifically for only one platform, or where you specifically want to build different versions for different platforms (although you can still do this with .NET MAUI), but most of the time it makes sense to use a cross-platform framework and build a single app that can be deployed to multiple target platforms.

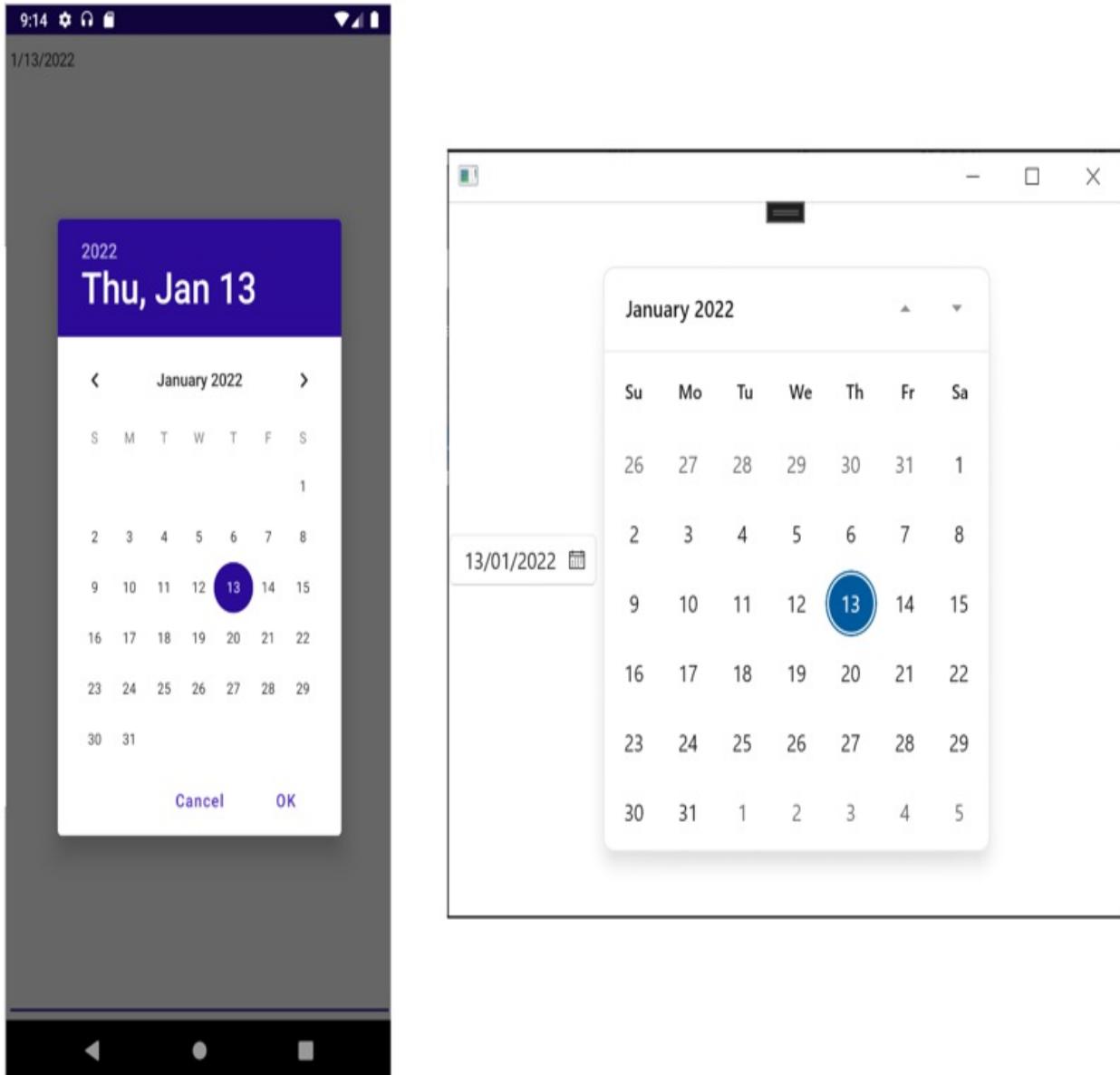
A cross-platform developer today has the luxury of choice in this respect. A popular approach is to build a web app and wrap it in an installable binary. Options for this include Ionic for Angular apps or Electron for anything web-based. This is an attractive option for some people, but while this can still give you full access to native platform APIs, it does have some limitations. Chief among these is that you are still using a web view to render and run your code, which carries with it all the performance and threading limitations of a web app.

The alternative approach is to use a single code base that can be built as a native app for each target platform. This is the approach used by .NET MAUI, as well as some other options such as React Native or Flutter, and it provides several advantages. These include multi-threading and other performance benefits, but also full access to all native platform APIs, in the case of .NET MAUI, guaranteed on the day of release. With web-app wrappers you are often dependent on plugins to provide this functionality, and there are no guarantees that the features you need access to are available.

The key distinction between .NET MAUI and other frameworks in this category is that the UI you build in .NET MAUI is an abstraction of the platform's native UI. This means that when you build an application in .NET MAUI, when it runs on iOS it looks like an iOS application, and when it runs

on Windows it looks like a Windows application, and the same for the other supported platforms.

Figure 1.6 A .NET MAUI DatePicker running in the same application on Android (left) and Windows (right). Both are running from the same code, using a DatePicker control from .NET MAUI, without any additional modification required to make them feel part of the platform they are running on.

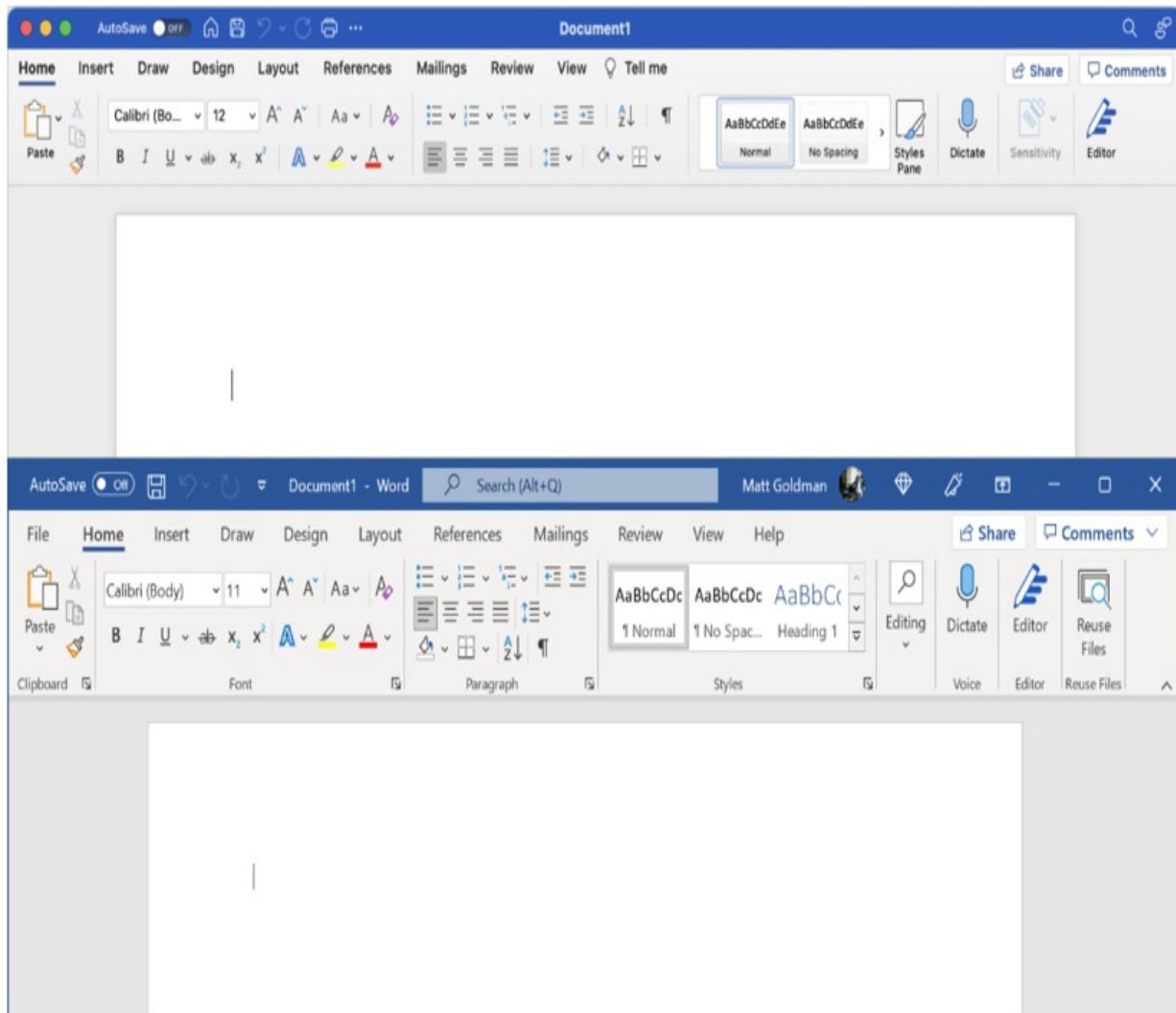


Of course, you can customize your UI to not use the native look and feel and use a fully custom UI that looks the same no matter where you run it. And this is the approach many people prefer. This approach takes no more effort

in .NET MAUI than in any of the other options. However, building an application that looks *consistent*, but not *identical*, on each platform, while remaining consistent with the platform too, requires no extra effort at all. To achieve this with React Native or Flutter would require multiple implementations of the same control – one for each platform.

This “consistent, but not identical” approach is the preferred way of building applications for many vendors. Let’s look at Microsoft Word as an example.

Figure 1.7 Microsoft Word running on macOS (top) and Windows (bottom). Each version remains consistent with its platform while retaining the product’s brand and UX.



On macOS, MS Word *looks* like a macOS application. On Windows it *looks*

like a Windows application. But both are consistent and the branding as well as navigation and UX are as familiar and comfortable to a user of one platform as they are to a user of another.

The truth is, though, all the available options are good. They are well established, mature, and have their share of supporters and detractors. Some provide a better approach to some aspects of cross-platform software development, while others provide a better approach in other areas. The decision to choose one framework over another will likely be most influenced by which is most squarely in your comfort zone. This could be influenced by several factors, but it's likely that the key factors influencing this decision will be who the primary backer of the framework is, and what the primary development language is.

If you love Microsoft and are an experienced C# developer, .NET MAUI is the obvious choice for you. If you're a Google fan and are comfortable with Dart, Flutter will likely be your first choice.

We're lucky to have so many available options. With the advent of NodeJS, a JavaScript developer, previously confined to web UI development, can now build a full-stack application in their favorite language. You may not think it's the best fit, but there are advantages to be gained from consistency across the stack. But, more importantly, they have a choice.

As .NET developers, we have this choice too. It's a great time to be a .NET developer. You can write server and cloud applications in ASP.NET, you can write web apps with Blazor, and you can write native desktop and mobile apps with .NET MAUI. And better yet you can share code between all of them.

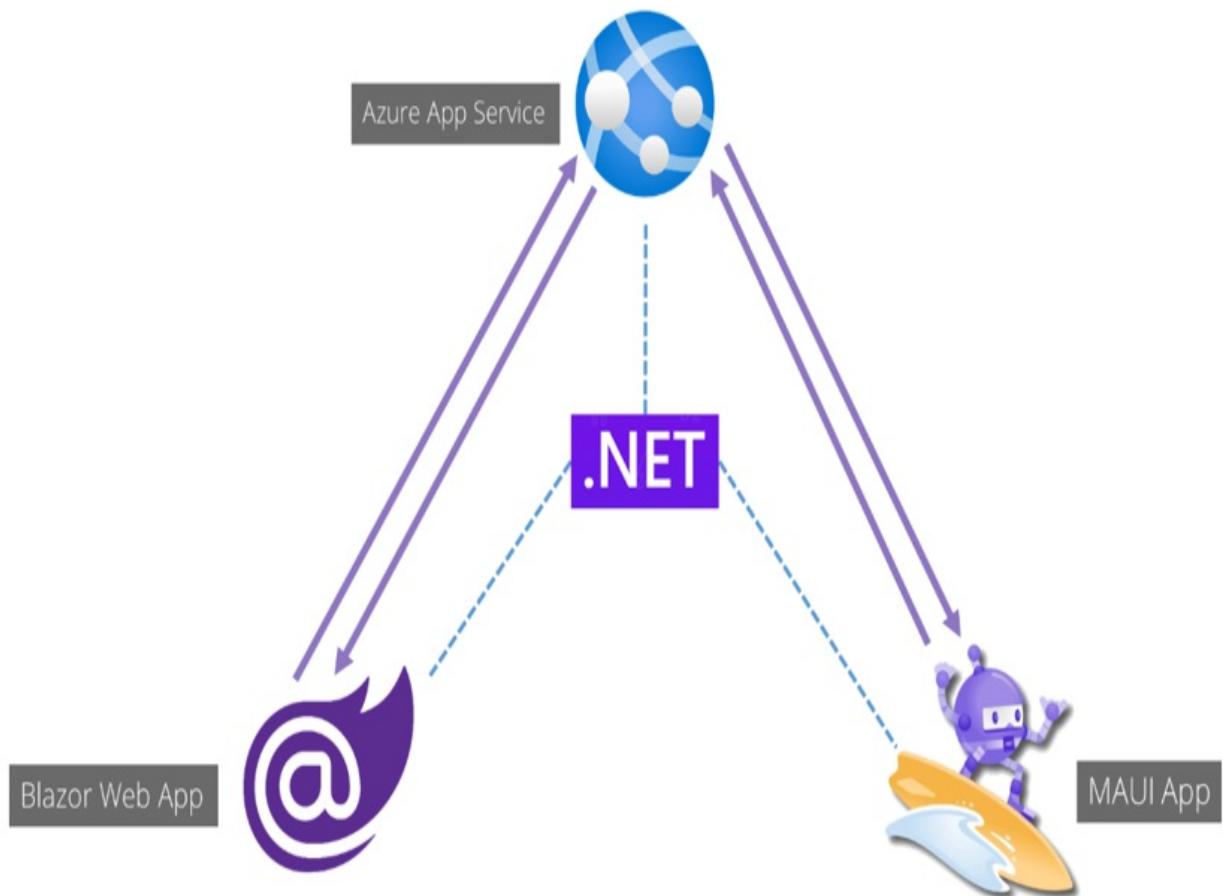
1.3 .NET MAUI and the .NET ecosystem

Writing a cross-platform application in .NET MAUI is a great option for .NET developers. You can use your favorite language and developer tools, and continue to use your skills and experience that you have built up as a .NET developer. You have access to all the same resources you use in other .NET projects, including your existing support networks, packages, and

patterns (although there are some new patterns to learn for mobile development too).

As a .NET project many, if not all, of your favorite NuGet packages are available to use in your .NET MAUI apps (whether these packages are suitable for mobile development is another matter – some are not), as are many that are specifically tailored to mobile and cross-platform UI development.

Figure 1.8 You can build full-stack cloud, web and desktop/mobile applications with all the components sharing a single code base.



If you are building a full-stack solution, you get the benefit of being able to share code between the different layers. This may not always be applicable, as different layers usually have very different responsibilities, but it's an excellent option to have. For example, if you are building a chat app with an

ASP.NET WebApi with SignalR running in Azure, a Blazor web UI, and a .NET MAUI mobile and desktop UI, you can share the logic and connectivity that links the UI to the WebApi across both your Blazor and .NET MAUI apps and in some cases use the same NuGet packages in both your client and server applications. If you make changes to your API, you can update the client code once and have the change automatically reflected across all your client UI applications.

As mentioned in section 1.1, with .NET MAUI now being a core part of .NET and a workload rather than an SDK, you also get to use all your familiar development tools. If it works with .NET, it works with .NET MAUI. This includes your favourite developer tools: Visual Studio (Mac or Windows), VS Code and the .NET CLI (although Visual Studio will provide a first-class experience); any build or DevOps tools; nuget packages; and anything else you can think of. The various components that make up your .NET developer experience, the .NET ecosystem as a whole, are at your disposal as a .NET MAUI developer.

.NET MAUI isn't an add-on, it *is* .NET. That means if it works with .NET, it works with .NET MAUI – and this counts for your skills as much as any other tool.

1.4 .NET MAUI Development Paradigms

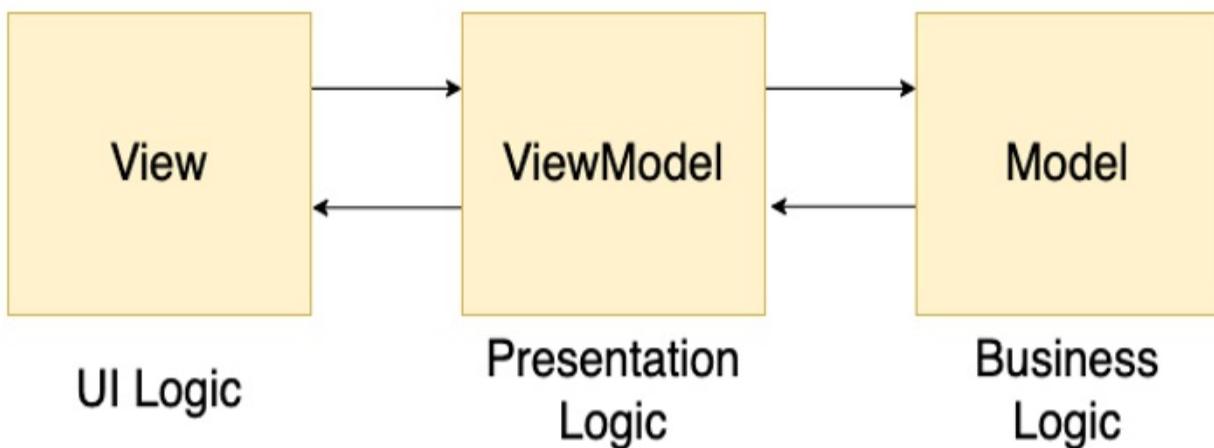
XAML (eXtensible Application Markup Language) is the de-facto choice for building applications in .NET MAUI. As mentioned in the introduction, Microsoft created XAML for WPF, but it has since also been used for Silverlight (in fact Silverlight was specifically a XAML renderer plugin for web browsers), Windows Phone, UWP and Xamarin.Forms, and now .NET MAUI too.

XAML is a good choice for most people. It's an XML based markup language for defining a UI, the same as HTML, or the plain XML used in Kotlin for building Android UIs, so is usually comfortable and quick to learn for people coming from Angular, plain HTML or Android, and especially people with XAML experience (WPF or Xamarin.Forms). The different 'flavors' of XAML can sometimes trip people up, there are subtle differences

between WPF, Xamarin.Forms and .NET MAUI XAML, but these differences are very quick to learn, and the excellently overhauled XAML Intellisense that you get with Visual Studio 2022 makes it even easier.

Complementary to XAML is the Model-View-ViewModel (MVVM) pattern. With MVVM, your XAML defined UI is called the View. The View consists of anything required by the app to display things on screen according to your design, including UI controls as well as any code required to change how they are displayed. The ViewModel represents the state of your View, and contains logic for interacting with the Model. The ViewModel responds to events in the View and passes data to the Model, and changes the state of the View when the Model provides information that requires the View state to change. The Model, consisting of objects and services that represent the problem that your app solves, contains business logic. The MVVM pattern is covered in more depth in chapter 7.

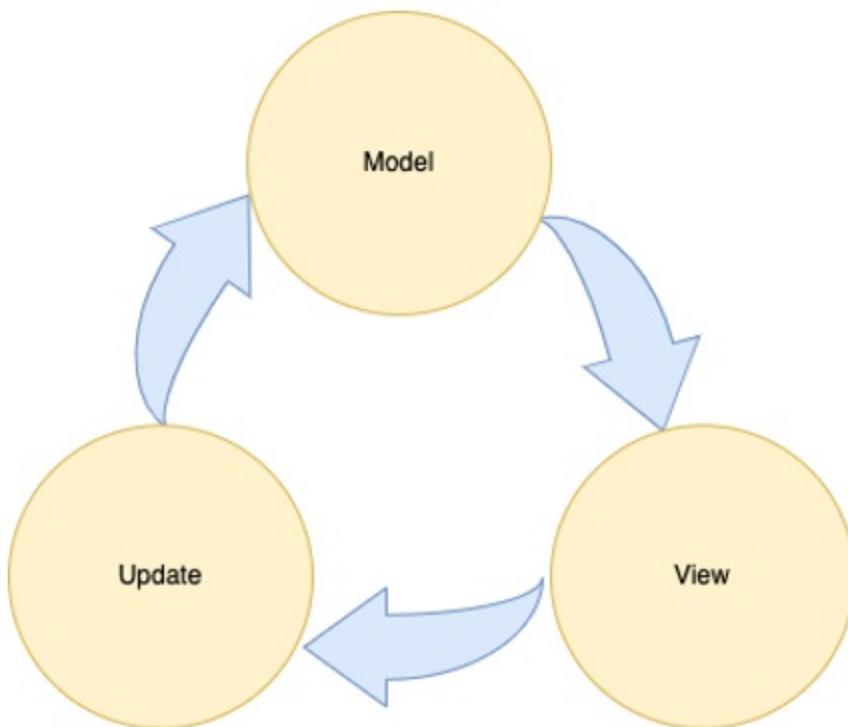
Figure 1.9 The MVVM pattern. Your Model is the representation of the problem your app solves, and consists of entities and services. Your ViewModel represents the state of your view, and contains logic for updating the model in response to changes in the View (user input) and for updating the View in response to changes in the Model. Your View contains logic for displaying your app on screen, including definitions of the UI as well as any logic for changing how the UI is rendered.



But XAML isn't your only choice. You can also declare your UI in code rather than in markup. This means declaring an instance of the class representing the UI control or view you want to display on screen and specifying its properties. Some people prefer this approach, but personally I

prefer to define my UI in markup; probably because the code approach evokes memories of drawing buttons on screen by specifying the coordinates to draw a black line for the top and left borders and a white line for the bottom and right borders (and then invert them when the user clicked on the button). I didn't have access to a UI library back then and had to draw all my controls by hand.

Figure 1.10 In the MVU pattern, the Model represents the whole-of-application state. Changes in the Model are sent to the View, which changes what is displayed when the Model is updated. Changes in the View are dispatched to an Update function, which will generate a new Model.



In addition to declaring your UI in code, you can also use the Model-View-Update (MVU) paradigm instead of MVVM (note that at the time of writing MVU support is experimental in .NET MAUI). MVU, also known as the Elm Architecture, differs from MVVM in two key ways. These are that the Model is immutable and that data flows in one direction only. This means that you can't change the Model in response to UI changes (user input), because this would violate both of these rules. Instead, changes in the View flow to an Update, which generates a new Model. The View then changes in response to the new Model.

MVU will be familiar to people coming from a native iOS development (Objective-C or Swift) background or React developers (the React Virtual DOM is a version of the MVU pattern). In .NET MAUI you need to bring in a library to support this pattern (Comet for C# and Fabulous for F#). In Xamarin.Forms, the legacy technology that .Net MAUI has evolved from, there was a tighter coupling between the Xamarin.Forms API and the underlying platforms, providing these kinds of patterns was more difficult. In .NET MAUI, abstractions provide a clean separation between the layers of the model (shown in figure 1.2), allowing different implementations to be brought in at any layer. This makes these MVU libraries ‘first class citizens’ in the .NET MAUI ecosystem, and they are fully endorsed by Microsoft.

MVU is not covered in this book, as with limited space the focus is on the core way of doing things. But if you are interested in MVU, there are plenty of resources online and a thriving community of .NET MVU enthusiasts.

.NET MAUI also gives you another option altogether, though, and that is to build your UI using Blazor. Blazor is a single-page application (SPA) framework created by Microsoft, that let’s you build applications that run client-side in web browsers using .NET rather than JavaScript or TypeScript. To learn more about Blazor, check out *Blazor in Action* by Chris Sainty, Manning 2022.

This approach may seem similar to the web app wrapping solutions mentioned in section 1.2, and in many ways it is. It lets you write a web app in a SPA framework (Blazor) and use a wrapper (.NET MAUI) to bundle that web app into an installable binary executable targeted at multiple platforms.

The key difference with .NET MAUI Blazor, though, is that where it uses a web view to render the UI, just the same as something like Electron, Ionic or Cordova, the C# code you write in a .NET MAUI Blazor app is run as .NET managed code, just like in a XAML MAUI app, rather than being run by the scripting engine in the web view, which is what you get with a web wrapper like Cordova, Ionic or Electron. Additionally, with a .NET MAUI Blazor app, you get access to all of the platform APIs that are exposed via .NET abstractions (as shown in figure 1.2).

With .NET MAUI, you can build a full-stack application using .NET at every

layer. You can use ASP.NET Core for your API, Blazor for your web app and .NET MAUI for your mobile and desktop clients. If you choose to use .NET MAUI Blazor, you can even share UI between web, mobile and desktop by placing your views in a Razor class library to share Blazor compatible UI across different projects.

.NET MAUI with Blazor may be a good option in a lot of cases. However, for the purposes of this book, we will be focusing on .NET MAUI development with XAML and the MVVM pattern. This is because the purpose of this book is to teach you .NET MAUI, not Blazor, and, as mentioned before, XAML with MVVM is the de-facto choice for .NET MAUI apps. Even if you do go on to learn other approaches, to fully understand .NET MAUI, you should learn this approach first.

1.5 Summary

- .NET MAUI is a cross-platform, write once, run anywhere (WORA) UI application platform. You can build just one .NET MAUI app and it will run on multiple platforms without further modification.
- You can write native apps with .NET MAUI. .NET MAUI apps are native apps.
- You can build apps in .NET MAUI that have functional, performance and security advantages over web apps.
- You can use the entire .NET ecosystem to build .NET MAUI apps. This includes all your favourite NuGet packages, as well as your existing skills as a .NET developer.
- You can write .NET MAUI app UIs in XAML, C#, F# or Blazor (we will be using XAML in this book).

2 Build a .NET MAUI app

This chapter covers:

- Introducing Visual Studio for macOS and Windows and the .NET CLI; the tools you will use to build .NET MAUI apps.
- How to create a new cross-platform mobile and desktop app with .NET MAUI app.
- How to run your .NET MAUI app and see changes in real-time with Hot Reload.

I work with .NET developers all day every day. Most of them are full-stack developers and work with a web UI framework like Angular or React. But I often hear them say things like “I don’t know mobile development” or “I don’t know native desktop development”. This is a misconception and couldn’t be further from the truth.

Any .NET developer can build mobile or desktop UI apps with .NET MAUI. There’s a small learning curve to get to grips with some of the UI- and markup-specific syntax and the design patterns – and that’s what this book is for. Anyone with prior experience with a web UI framework (especially Angular) should feel very comfortable working in .NET MAUI; although prior experience is not necessary.

In this chapter we’ll see just how easy it is for .NET developers to get started building mobile and desktop UI apps with .NET MAUI.

2.1 Say ‘Aloha, World!’ with .NET MAUI

In this section, we’re going to build our first .NET MAUI App: Aloha, World! We’ll look at the tools and templates available to developers for building .NET MAUI apps and talk about some of the pros and cons of each approach.

Which approach you choose is up to you and will be entirely dependent on

your comfort, experience, and workflow. Whichever approach you prefer, I recommend walking through both using Visual Studio (for either Windows or macOS, depending on what you have available) and using the command line so you at least gain some familiarity with both approaches.

Which option should I choose?

It's up to you to decide whether you would prefer to use Visual Studio or the .NET CLI. Whether you choose Visual Studio for Mac or Visual Studio for Windows will depend on what hardware and operating system you have available.

Visual Studio, on both macOS and Windows, has some sophisticated development features that you don't get with the CLI. For example, the XAML Live Previewer in Visual Studio can give you real time feedback on design changes (see section 2.4 later in this chapter for more information). Visual Studio also has powerful Intellisense, and now Intellicode, code-completion features that can be indispensable when writing .NET MAUI apps, not to mention mature testing and debugging features.

Another advantage of using Visual Studio is that it's easy to choose your target platform; you simply use a dropdown built into the Run button. Using the .NET CLI is a little bit more complicated, and perhaps more importantly, the .NET CLI doesn't support targeting Windows (you can still build and run .NET MAUI apps on Windows using a command prompt, but you have to use MSBuild rather than the .NET CLI).

Visual Studio is a great option for .NET developers and, except where I'm explicitly demonstrating the .NET CLI, will be used in the examples and screenshots throughout this book (although all the samples will work fine with the .NET CLI too). But Visual Studio is not without its detractors. For many people, Visual Studio is *too* powerful, and some prefer a lightweight alternative. This is where the .NET CLI comes in.

The primary benefit of any GUI is discoverability; it's much easier to click around in Visual Studio than it is to delve into the documentation of the .NET CLI to figure out how to do this or that. But while a CLI may have a steeper learning curve, there's no question that an adept CLI user can see significant

productivity and efficiency gains over their GUI-using counterpart.

Entering the following commands takes a second or two:

```
mkdir AlohaWorld  
cd AlohaWorld  
dotnet new blankmaui
```

It takes the .NET CLI about 0.4 seconds to build a new .NET MAUI app from the template; the equivalent in Visual Studio can take significantly longer. As well as the efficiency gains, using the .NET CLI may seem more familiar to developers coming from other frameworks. The Angular CLI, for example, has commands with many analogs in the .NET CLI.

If you choose to use the .NET CLI, you will still need to use software to edit your code. Visual Studio Code, a text editor with developer-focused features such as syntax highlighting, is a popular choice among many developers, and has a rich community-supported extension ecosystem. But you could use anything you like – essentially a .NET MAUI app is a collection of text files, at least until you execute `dotnet build` to compile them, so any software capable of editing text files will work. But using a developer centric text editor like Visual Studio Code will vastly simplify your development experience.

Personally, I like to use a combination of both Visual Studio and the .NET CLI. I find the .NET CLI much quicker for some tasks, while I find Visual Studio indispensable for others (such as debugging). Most of my .NET MAUI development takes place in Visual Studio, but you should find the most comfortable mode of working for you. Whichever you choose, it's worth at least learning the fundamentals of all the tools at your disposal.

Ensure you have the .NET MAUI workload installed, either via the Visual Studio installer or the .NET CLI. We're also using a blank .NET MAUI project template that you need to install separately. If you are unsure about whether you are ready to proceed, see Appendix A for getting set up with all the right tools. Once you're ready, dive in to create your first .NET MAUI app.

2.1.1 Visual Studio 2022

Visual Studio is the primary first-party tool provided by Microsoft for developing apps with .NET MAUI. The latest version, 2022, is the minimum required version for .NET MAUI, and has built in support for a lot of features that make developing .NET MAUI apps a lot more efficient. We'll see some of these features, like XAML Live Preview (covered in Chapter 3) and .NET Hot Reload (covered later in this chapter) as we progress through the book.

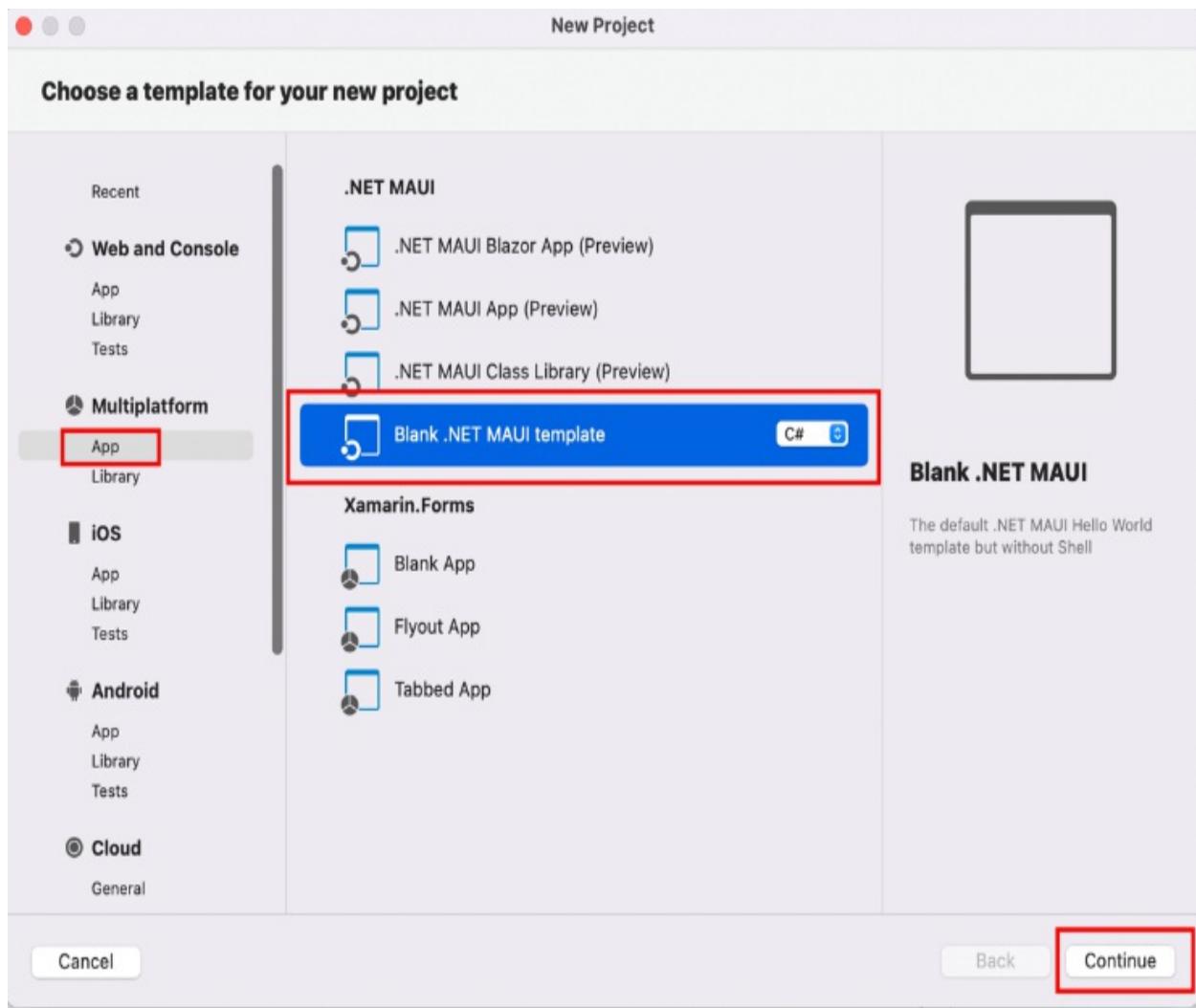
Visual Studio is available for Windows and macOS, with different editions ranging from the free Community edition through to the top tier Enterprise edition. .NET MAUI works with all of these, and everything covered in this book can be done with the free Community edition, although if you already have a Visual Studio subscription, you can of course use it for .NET MAUI. For more information, see Appendix A.

Mac

To get started, open Visual Studio 2022 and click on New.

The next screen will ask you to choose a template. Scroll down the list on the left until you get to the Multiplatform section. In here click on App to bring up the .NET MAUI templates. Choose Blank .NET MAUI template from the list. With this template selected, click on the Continue button.

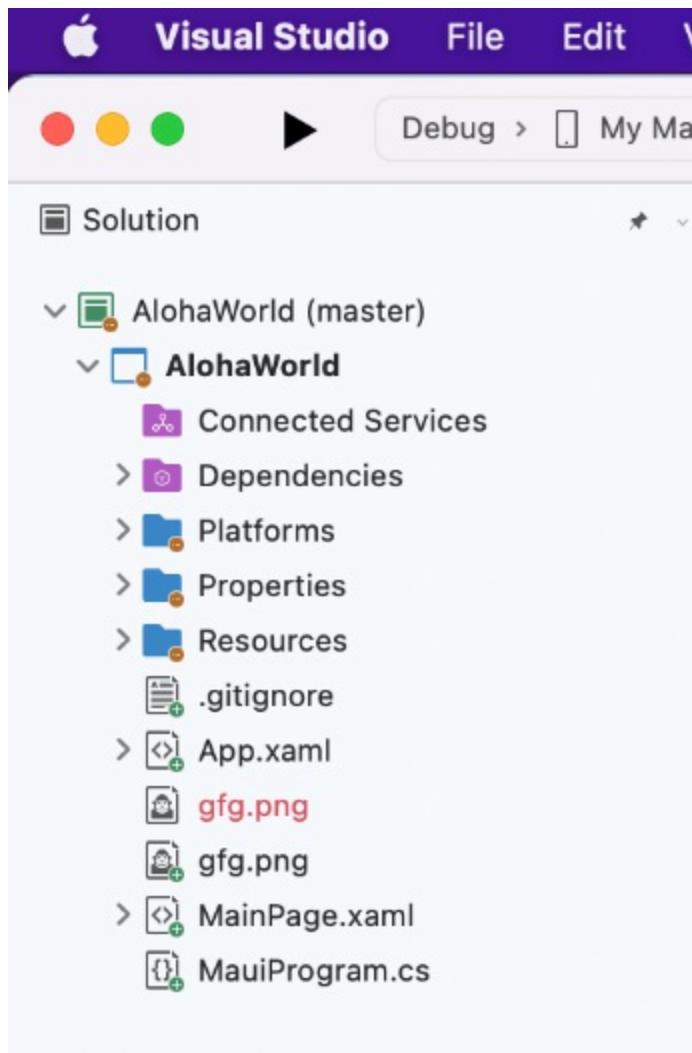
Figure 2.1 Find App under the Multiplatform section from the categories on the left. This will show the available .NET MAUI templates; choose the Blank .NET MAUI template to generate a new blank app..



Enter AlohaWorld as the project name. By default, the solution name will automatically be set the same name as the project name. Leave this as it is for now; it makes sense to have these be different when you're building complex, multi-project solutions, which we're not doing here. Visual Studio will suggest a default location to save the app; feel free to select a location of your choice. You'll also be given some options for version control; we're not using version control for this app, but you're a git user and want to use it for this app, feel free to leave it ticked if you wish.

Visual Studio will take a short amount of time to create the new solution for you based on the .NET MAUI App template. When complete, you should see a collection of files and folders in Solution Explorer, matching what you see in Figure 2.2

Figure 2.2 The files in the newly created AlohaWorld .NET MAUI app. We'll talk about these files in section 2.3.



We'll look at these files in section 2.3 and see how .NET MAUI starts and launches an app. For now, your .NET MAUI app has now been created from the template and is ready for you to start working on.

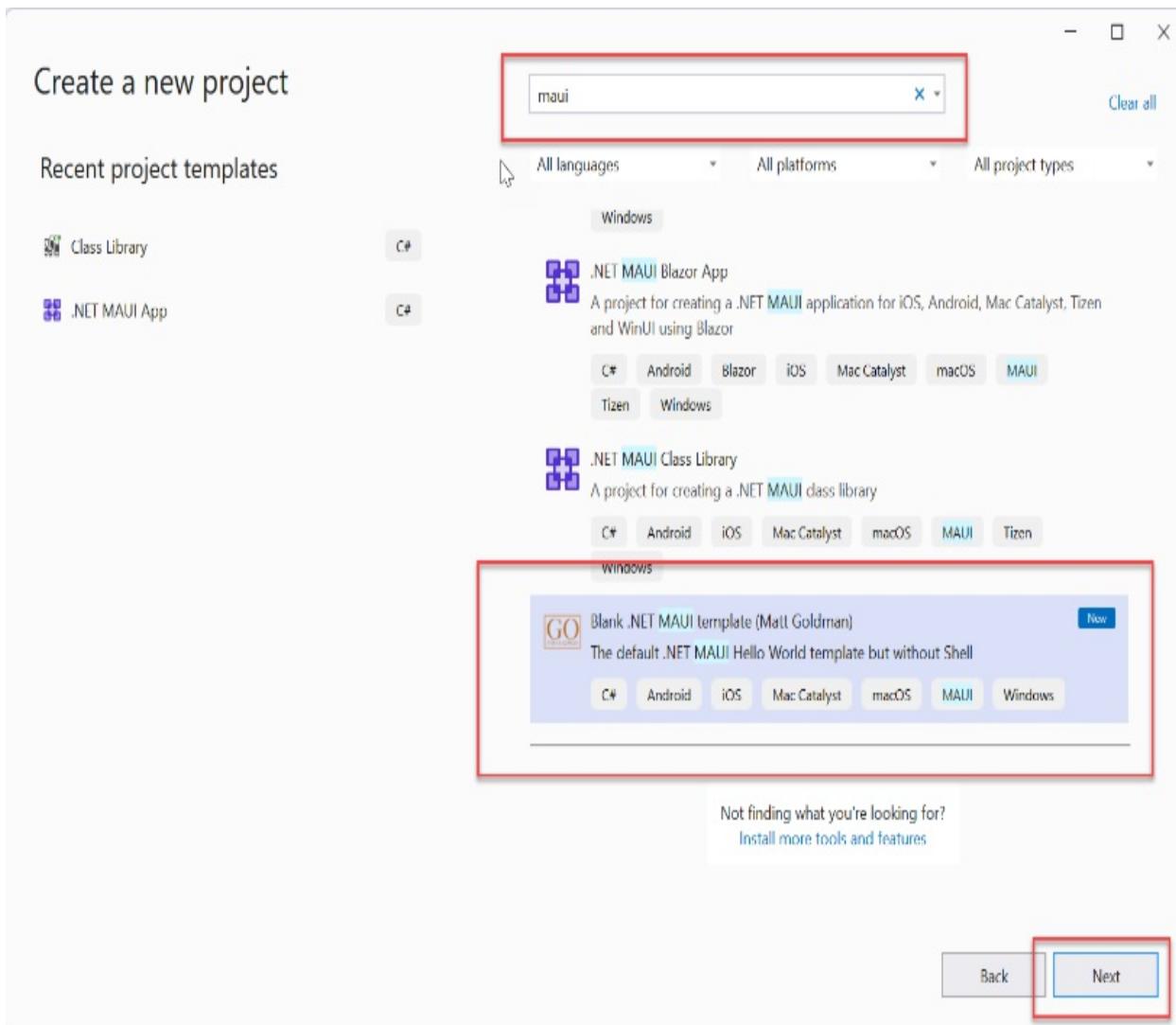
Windows

To get started, open Visual Studio, and click on Create a new project.

Visual Studio will show you a list of available project templates, with your most recently used templates on the left, and all project templates on the right. Select the Blank .NET MAUI template from the list. If you can't see it,

you can use the filtering features. Use the dropdown boxes to filter by language, platform or project type; although I often find that these don't limit results in a meaningful way. To make it easier, you can enter the search term 'maui' in the search box to filter the available templates to choose from (see Figure 2.3). With Blank .NET MAUI template selected, click on Next.

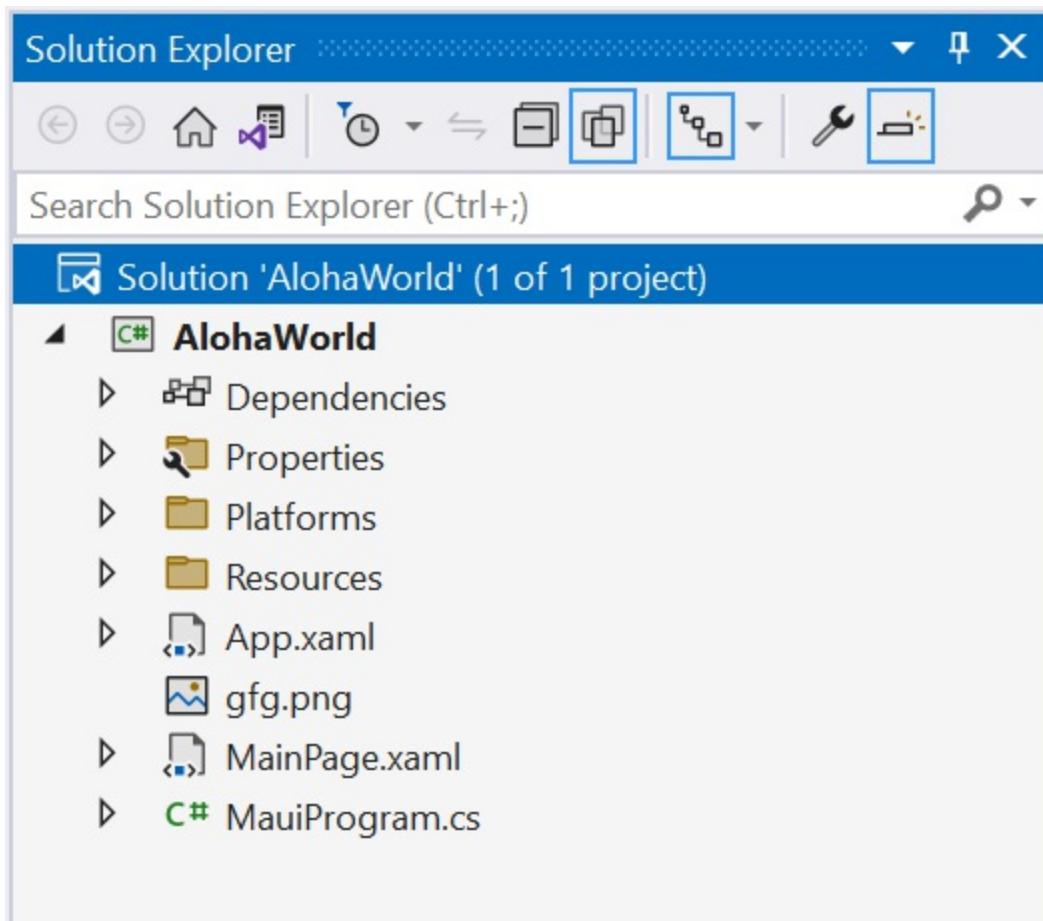
Figure 2.3 You can filter project templates in Visual Studio with the language, platform or project type dropdowns, but the quickest way to find what you're looking for is with the free text filter. In this example, we've entered 'maui' as a search term and narrowed the list of project templates to .NET MAUI projects.



Choose a folder to save your project into (or use the default location) and enter 'AlohaWorld' as the project name. Then click Create.

Visual Studio will take a short amount of time to create the new solution for you based on the .NET MAUI App template. When complete, you should see a collection of files and folders in Solution Explorer, matching what you see in Figure 2.4

Figure 2.4 The AlohaWorld solution in Visual Studio Solution Explorer. We'll talk about these files in section 2.3.



We'll look at these files in section 2.3 and see how .NET MAUI starts and launches an app. For now, your .NET MAUI app has now been created from the template and is ready for you to start working on.

2.1.2 .NET CLI - Overview

The .NET CLI is a comfortable and familiar tool for most .NET developers. Some people prefer to remain strictly in the GUI, that's perfectly fine. The

.NET CLI is easy to use, though – the only thing you really need to learn is how to use the interactive help. The commands are deliberately discoverable, and once you learn them, they can make your development experience much more efficient.

As .NET MAUI is a .NET workload rather than an external package, the project templates work the same way as all other .NET templates and accept the same inputs and switches. These are covered in the Microsoft documentation and are also discoverable via online help, which can be accessed by adding `--help` after any dotnet command.

Let's start by looking at the available templates. Open your command prompt of choice and enter `dotnet new --list`. You should see a list of available templates available to the .NET CLI tool, and conveniently the .NET MAUI templates are at the top. Table 2.1 below provides a brief explanation of these templates and their use cases.

Table 2.1 A summary of the .NET MAUI Templates

Template Name	Short name	Description
.NET MAUI App	maui	This is the main template used for creating new .NET MAUI apps and is the one we will use throughout this book
.NET MAUI Blazor App	maui-blazor	This template is used to create a new .NET MAUI app that uses Blazor to define its UI
.NET MAUI Class Library	mauilib	This template creates a new class library for sharing code between different .NET MAUI projects

.NET MAUI ContentPage (C#)	maui-page-csharp	This template creates a new .NET MAUI application page (see section 1.3), with UI defined declaratively in C#
.NET MAUI ContentPage	maui-page-xaml	This template creates a new .NET MAUI application page with UI defined in XAML markup (and a corresponding C# code behind file)
.NET MAUI ContentView (C#)	maui-view-csharp	This template creates a new .NET MAUI content view (a reusable UI component that can be used in .NET MAUI application pages) with UI defined declaratively in C#
.NET MAUI ContentView	maui-view-xaml	This template creates a new .NET MAUI content view (a reusable UI component that can be used in .NET MAUI application pages) with UI defined in XAML markup (and a corresponding C# code behind file)
.NET MAUI ResourceDictionary (XAML)	maui-dict-xaml	This template creates a new .NET MAUI resource dictionary in XAML. This allows you to define and name colours, styles and templates that can be referenced by name for reuse throughout your app.

Blank .NET MAUI App	blankmaui	This is an additional template that I provide. It is the same as the default template (<code>maui</code>), except the default template uses Shell (which we won't talk about until chapter 5). This template does not use Shell so should be easier to work with for the first few chapters.
---------------------	-----------	--

The .NET CLI has some conventions that make it easy to use. For example, if you use the `dotnet new` command with a template of your choice, without specifying any other parameters, the .NET CLI will create a new solution for you based on your selected template, using all the default options, and using the name of the containing folder as the solution name.

Overriding default values

You can specify additional command line parameters to the .NET CLI to override this default behavior. For example, you can specify a different output directory or a different solution name (or both) using the available command line switches. If we wanted to call our application `HelloWorld` instead of `AlohaWorld`, and build it in a directory called `Code` in the root of our hard drive, we could do so with the following command:

```
dotnet new blankmaui -n HelloWorld -o C:\code\HelloWorld
```

For more information about all the available command line switches, use the online help (`dotnet new --help`) or consult the documentation.

We're going to take advantage of this simplicity to create our brand-new .NET MAUI app.

2.1.3 .NET CLI in Action

Create a folder called `AlohaWorld`. You can use your operating system's file browser (Explorer on Windows or Finder on macOS), or you can use the

command line.

Then, using your command-line terminal of choice, navigate into that folder.

In the new folder you created, enter the command:

```
dotnet new blankmaui
```

and press enter. If the command executed successfully, you should see a message saying:

The template "Blank .NET MAUI template" was created successfully.

We can now examine the contents of the folder to see the files that have been created for us by this template. Use your operating system's file browser or the command line to show the contents of the folder (in Figure 2.5 we use the command line).

Figure 2.5 The .NET MAUI template generates solution files for us to use, seen here listed in the terminal

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

matty@GFG-DESK E:\code 2022-05-24 20:35:10
mkdir AlohaWorld

Directory: E:\code

Mode LastWriteTime Length Name
---- -- -- - - -
d--- 24/05/2022 8:35 PM AlohaWorld

matty@GFG-DESK E:\code 2022-05-24 20:35:17
cd ..\AlohaWorld\
matty@GFG-DESK E:\code\AlohaWorld 2022-05-24 20:35:19
dotnet new blankmaui
The template "Blank .NET MAUI template" was created successfully.
Warning: Your request could not be authenticated by the GitHub Packages service. Please ensure your access token is valid and has the appropriate scopes configured.

matty@GFG-DESK E:\code\AlohaWorld 2022-05-24 20:35:29
ls

Directory: E:\code\AlohaWorld

Mode LastWriteTime Length Name
```

You have now created a .NET MAUI app from the template and it is ready for you to start working on.

2.2 Run and debug your app

Now that we've created our first .NET MAUI app, let's see it in action! The following sections will guide you through the steps of running your brand-new .NET MAUI app depending on which development approach you're using.

Single Project Solutions

If you are coming from Xamarin.Forms, you'll notice some differences here. In Xamarin.Forms we had a project for each platform (e.g., MyApp for the shared logic and UI, MyApp.Android for Android, and MyApp.iOS for iOS), and would set the project for the target platform we wanted to run our app on as the startup project. In .NET MAUI, we have a single project solution, and

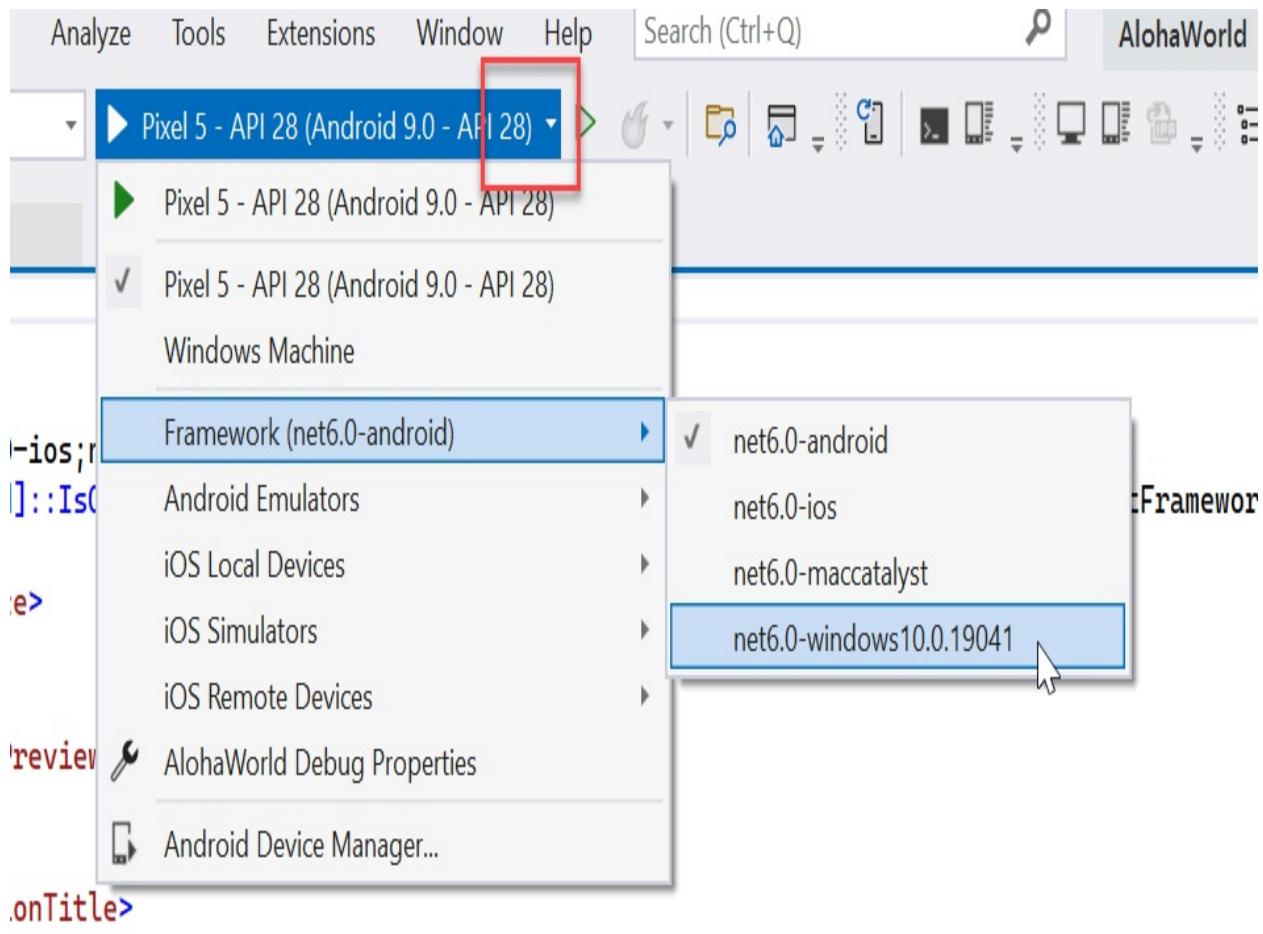
use multi-targeting to choose where we want to run it. We can of course add class libraries or other projects if we wish, but this is not necessary for targeting different platforms.

2.2.1 Visual Studio for Windows

The following steps will guide you through the process of building and running your .NET MAUI app. These steps show you how to choose Windows as your target platform and run ‘Aloha, World!’ as a Windows app. You can also run the app on Android or iOS from Visual Studio for Windows – for steps on how to do this, see Appendix A.

Use the drop down on the Run button in the toolbar. From the menu, choose the Framework sub-menu, and choose net6.0-windows[your windows build version] as the target (see Figure 2.6).

Figure 2.6 Use the drop to select a target platform

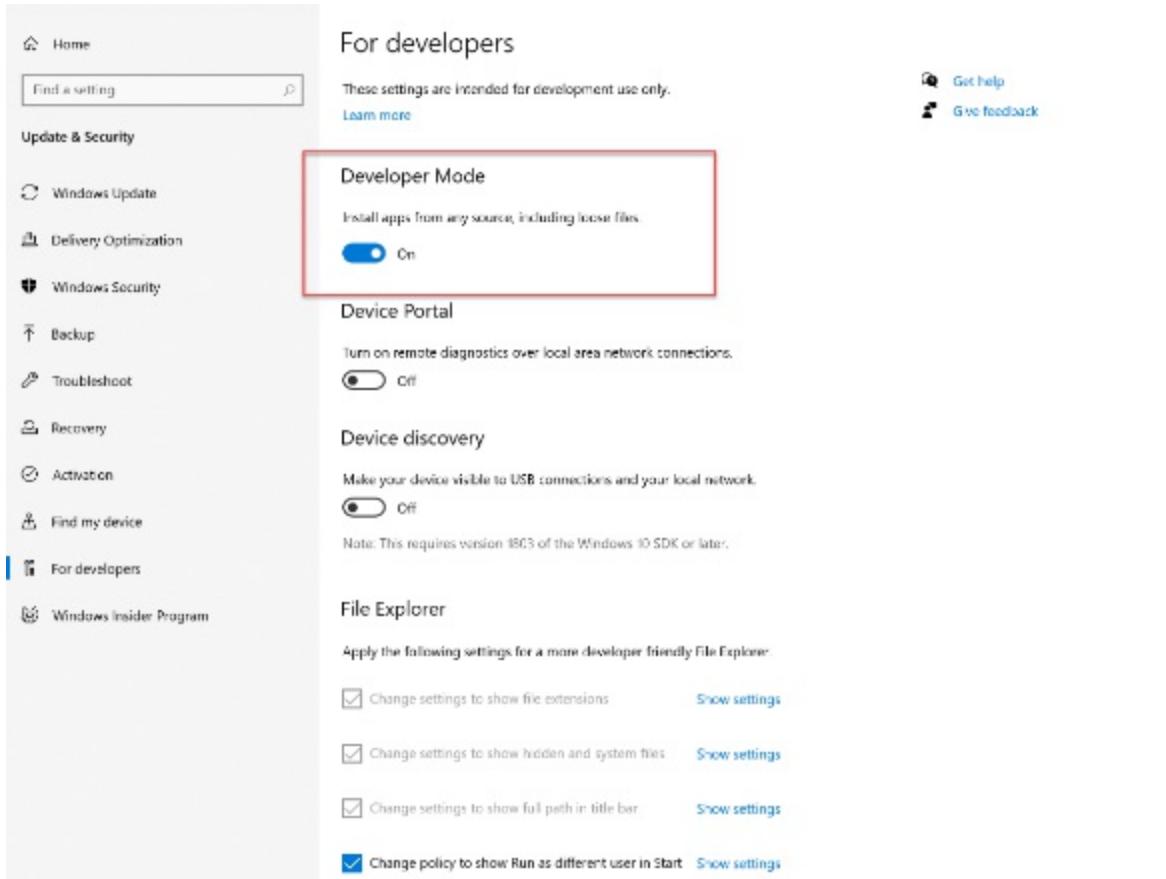


You can now run your app on Windows by clicking the Run button on the toolbar. You could also chose to run the app on Android, and we cover targeting different platforms in chapter 9.

Note that you will have to enable Developer Mode on Windows if you have not already. Developer Mode lets you run ‘unsigned’ apps on Windows. By default, executables that are not signed by a trusted authority are blocked from running. While you are developing apps in .NET MAUI, they will be unsigned, so Developer Mode is required, but you should consider disabling it while you’re not actively using it. This will help to keep your system secure

This can be done in the built-in Settings app (See Figure 2.7). Access this by hitting the Windows key and start typing Settings. It will appear in your search results, and you can press enter to open it.

Figure 2.7 Enable developer mode in Windows to run unsigned .NET MAUI apps



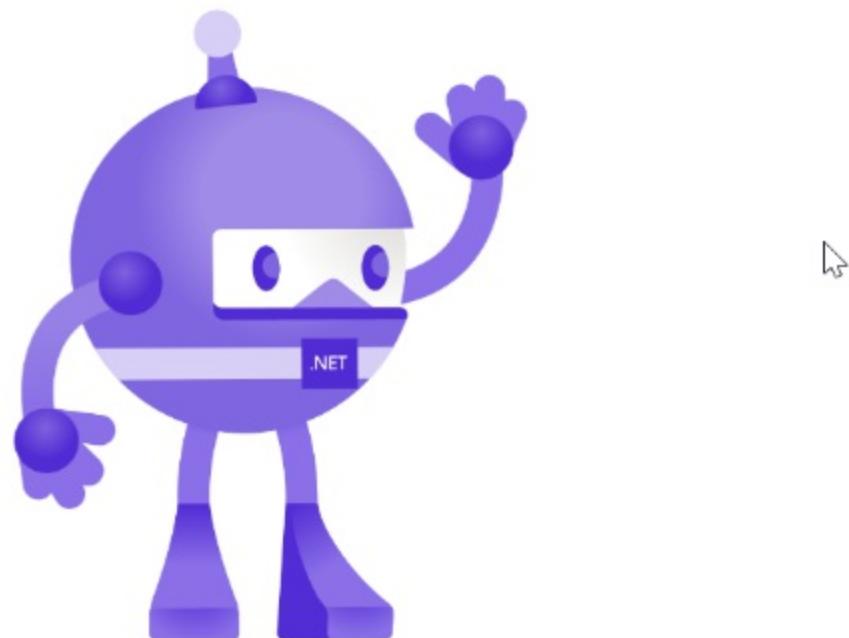
Now you should see your app running (see Figure 2.8). We'll go through what we're seeing here on screen in the next section, but, for now, go ahead and click the counter button and see what happens.

Figure 2.8 The AlohaWorld app running on Windows – click the counter to see changes

Hello, World!

Welcome to .NET Multi-platform App UI

Current count: 0

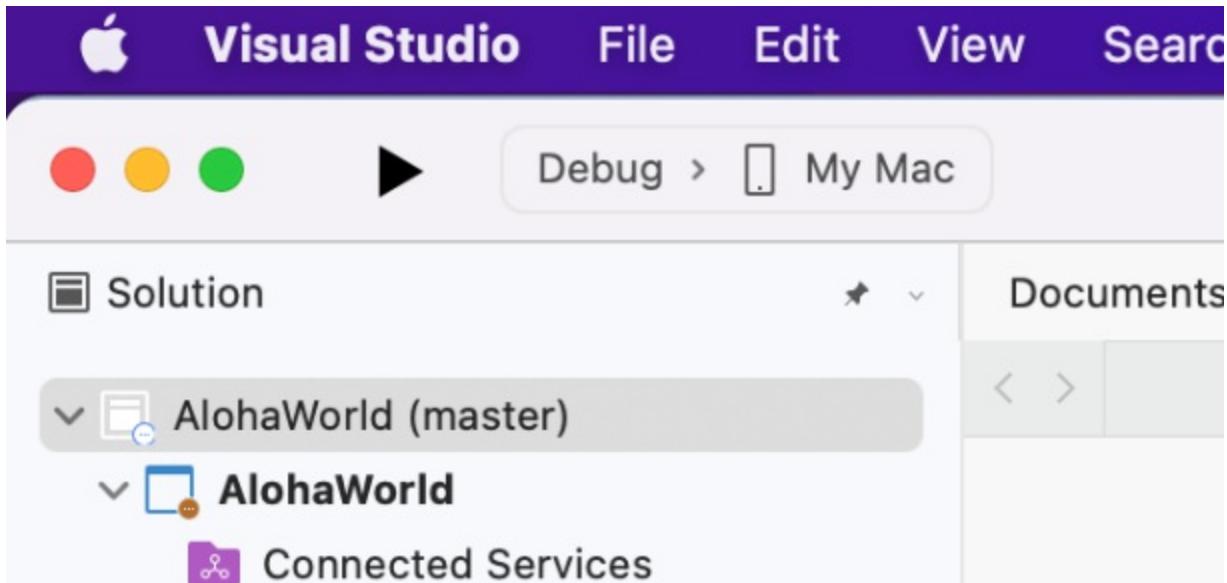


Have fun with your first .NET MAUI app that you have built and run!

2.2.2 Visual Studio for Mac

Visual Studio 2022 for Mac makes it easy to run and debug your .NET MAUI apps on macOS or iOS. Near the top-left of the screen, you should see a run button, and just to the right of it you should see the profile (Release or Debug) and then the target (by default this will say My Mac).

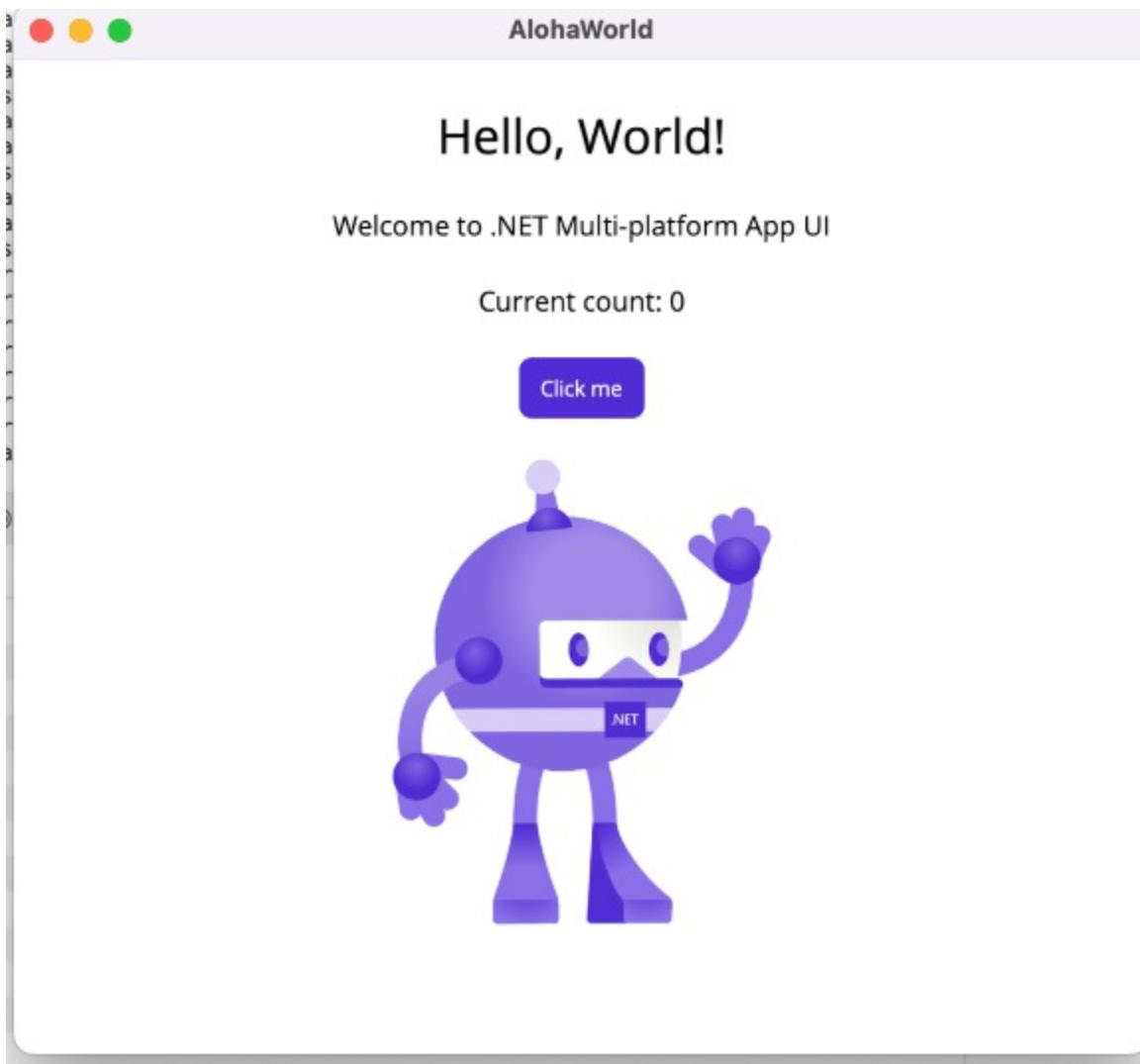
Figure 2.9 The run button in Visual Studio 2022 for Mac, with the profile and target shown next to it



Clicking on My Mac will drop down a list of all the target devices you can run your .NET MAUI app on. This will include the Mac you are running Visual Studio on, and any currently supported iOS device simulators. Later, in chapter 9, we'll see how you can target a physical iOS device too.

Leave My Mac selected and click the run button.

Figure 2.10 The Aloha, World! .NET MAUI app running on macOS



Now you should see your app running (see Figure 2.10). We'll go through what we're seeing here on screen in the next section, but, for now, go ahead and click the counter button and see what happens.

2.2.3 .NET CLI

Using your terminal console of choice, navigate to your solution folder and enter the run command corresponding to your target platform. The target platform is the platform on which you want to run your .NET MAUI app, not the platform you have developed it on. Use the reference guide below to choose the right platform target.

The run command is

```
dotnet build -t:Run -f:net6.0-[target platform]
```

The target platforms are:

Table 2.2 .NET MAUI target platforms

Operating System	Target Platform	Notes
macOS	maccatalyst	<p>Mac Catalyst is a bridge that lets you run apps built for iOS on macOS. .NET MAUI uses maccatalyst to run apps on macOS.</p> <p>You can only target macOS when developing on macOS.</p>
iOS	ios	<p>You can target iOS when developing on either macOS or Windows (although a macOS computer is required to publish .NET MAUI apps to the App Store).</p>
Android	android	<p>You can target Android when developing on either macOS or Windows.</p>

where is windows?

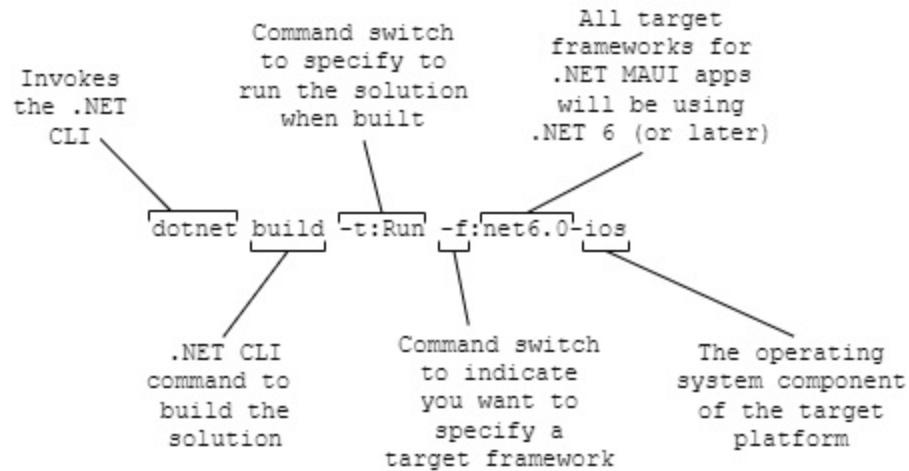
While Windows is a target platform for .NET MAUI, it's missing from table 2.2 because you can't build .NET MAUI apps for Windows with the .NET CLI. It's best to stick to Visual Studio, although if you can use MSBuild if you want to use a command terminal. Check Appendix A for instructions on running .NET MAUI apps at the command line with MSBuild.

So, to run your app on iOS, you would use the command

```
dotnet build -t:Run -f:net6.0-ios
```

Let's look at the components of this command so we can see how it works, and what it does (Figure 2.11).

Figure 2.11 Breakdown of the CLI command to run your .NET MAUI app



The first thing to notice here is the `dotnet` command. This is the name of the executable you want your operating system to run; in this case the .NET CLI. The second part, `build`, is the command we want the .NET CLI to execute. This tells the .NET CLI to build our project or solution, and the .NET CLI will expect a `.csproj` or `.sln` file in the current directory (we can use additional parameters to specify a different project if we wish).

The next part, `-t:Run`, tells the .NET CLI that we want it to run the solution after the build has completed. Finally, the `-f:` switch tells the .NET CLI that we are going to pass a framework parameter that we want it to use as the target platform. In the example in Figure 2.11, we have passed in iOS. All frameworks use `net6.0-` as the prefix, as .NET MAUI requires .NET 6 as a minimum (other options will be available for this prefix when newer versions of .NET are released).

Now that we understand how to use the `dotnet build` command to run our .NET MAUI app, go ahead and run the command to build and run the app on your target platform.

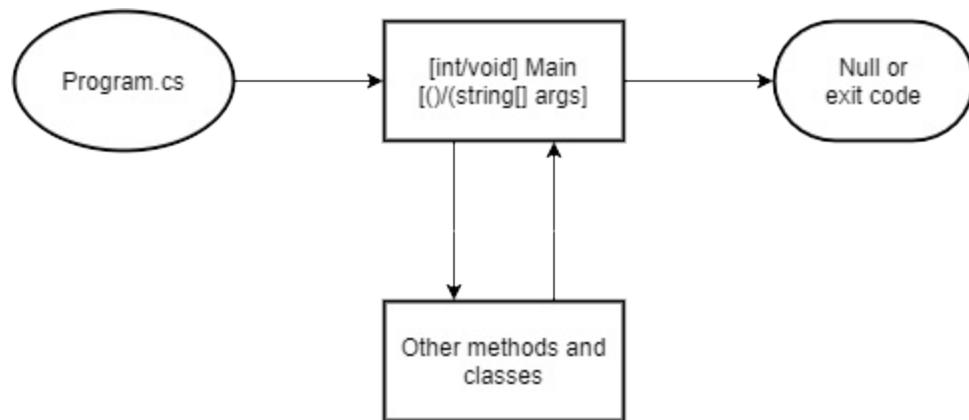
Congratulations, you have just built and run your first .NET MAUI app! Go

ahead and click the ‘Click me’ button to see the changes to the counter.

2.3 Anatomy of a .NET MAUI App

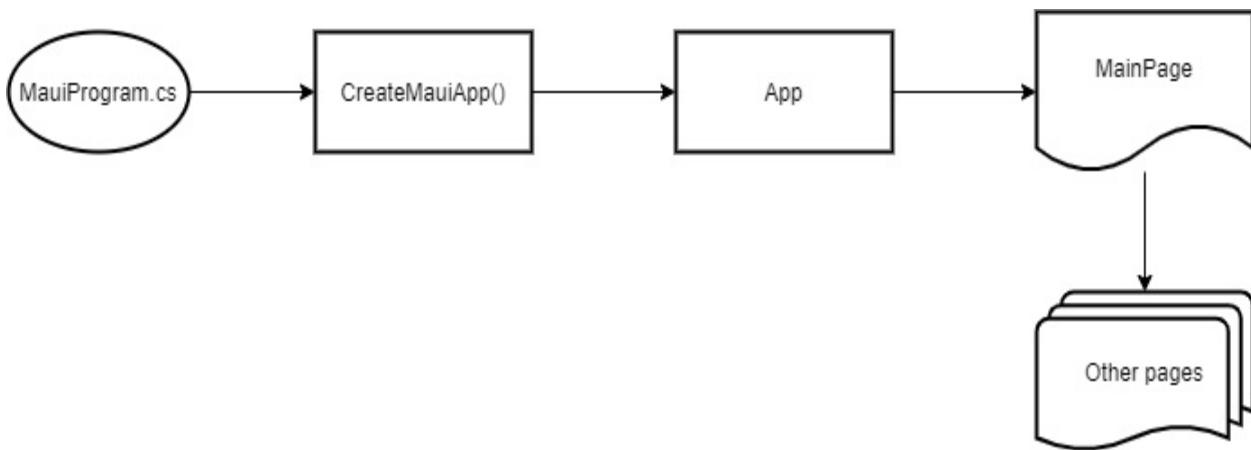
Now that the app is running, let’s look at how .NET MAUI runs our code, and how it differs from a normal C# program.

Figure 2.12 A regular C# program starts with the Main method in Program.cs and executes code until it exits, returning either null or an exit code



In a C# program, .NET usually looks for the *entry point*, which is a static method called `Main`, that returns either `int` or `void`, and has either no parameters or a single parameter of type `string[]` called `args` (see figure 2.12).

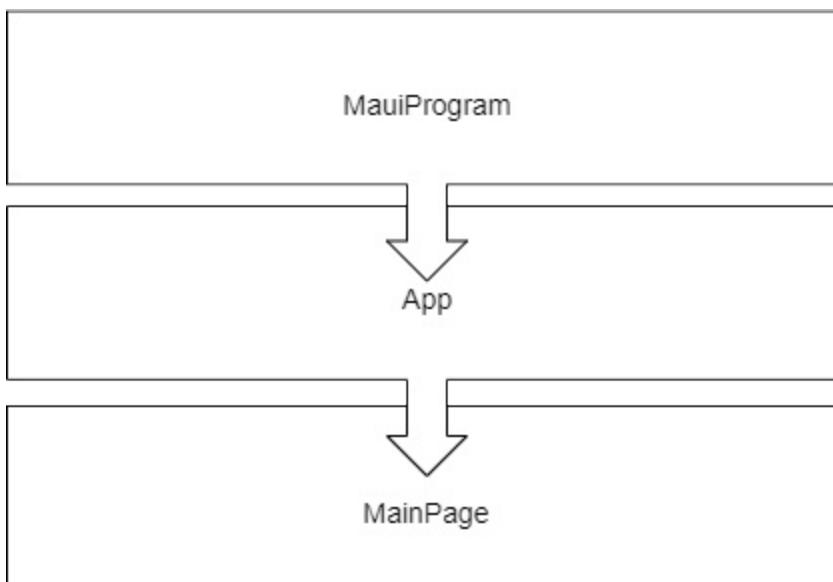
Figure 2.13 A .NET MAUI app starts with the CreateMauiApp method in MauiProgram.cs, and launches an instance of the App class, which displays a Page assigned to the MainPage property of the App class



In a .NET MAUI app, .NET expects a method that returns an object of type `MauiApp`, and this is in the `MauiProgram.cs` file that was created for us by the template. In here is a static method called `createMauiApp` that uses the generic hostbuilder pattern common across .NET 6, with a return type of `MauiApp` (figure 2.13)

A simplified version of this flow is shown in figure 2.14.

Figure 2.14 In a .NET MAUI app, `MauiProgram` is the entry point, which launches `App`, which displays `MainPage`



The `CreateMauiApp` method (in listing 2.1) has a return type of `MauiApp`, which is what will run our application. This static method builds this for us using a version of the .NET host builder, which ASP.NET or Blazor

developers will recognize.

An extension method is called on the hostbuilder called `UseMauiApp`, and is passed `App` as a type parameter. The `UseMauiApp` extension method expects a type that implements the `IApplication` interface, and the `App` class inherits the `Application` class, which implements this interface.

Listing 2.1 MauiProgram.cs

```
namespace AlohaWorld;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()#A
    {
        var builder = MauiApp.CreateBuilder();#B
        builder
            .UseMauiApp<App>();#C
            .ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
        });

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

The `App` class (listing 2.2) has a member called `MainPage`, and in the constructor of the `App` class we assign a value to this member of type `ContentPage` (which inherits the `Page` base class). .NET MAUI then displays that page to the user when the app has finished loading.

The type that we are instantiating is also called `MainPage`; although note that this is the name of the class, whereas in the `App` class, `MainPage` is a member of type `Page`. `Page` is a type in .NET MAUI that is used to define pages in our application. A page is a full-screen view, and a view is something that appears on screen. We will talk more about pages and views in Chapter 7.

The important point to note is that to start our app, we need to have an object

of type `Page` (it doesn't have to be called `MainPage`, it can be called anything you like, although it is called `MainPage` in all the templates), that we set as the value of the `MainPage` member in our `App` class. .NET MAUI will load the app and display this page to the user.

Listing 2.2 App.xaml.cs

```
namespace AlohaWorld;  
  
public partial class App : Application#A  
{  
  
    public App()  
    {  
        InitializeComponent();  
  
        MainPage = new MainPage(); #B  
    }  
}  
}
```

This is all you need to know to get up and running with .NET MAUI - but there is more to the story. On Windows, the process described above is accurate; however, on macOS, iOS and Android, there are specific entry points expected by the operating system SDKs to start an app, and .NET MAUI gives you instances of these that you can customise if you need to.

These are the `AppDelegate` class for macOS and iOS and the `MainActivity` class for Android. You can see these in Figure 2.15.

Figure 2.15 Platform specific entry points for iOS, macOS and Android in a .NET MAUI app kick off the regular program execution flow

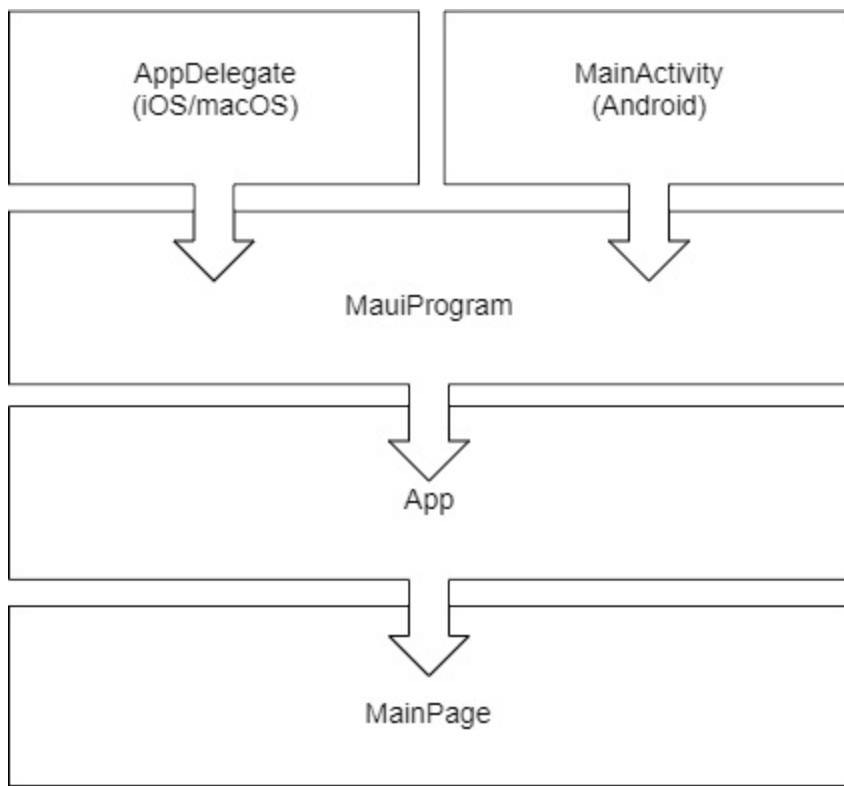
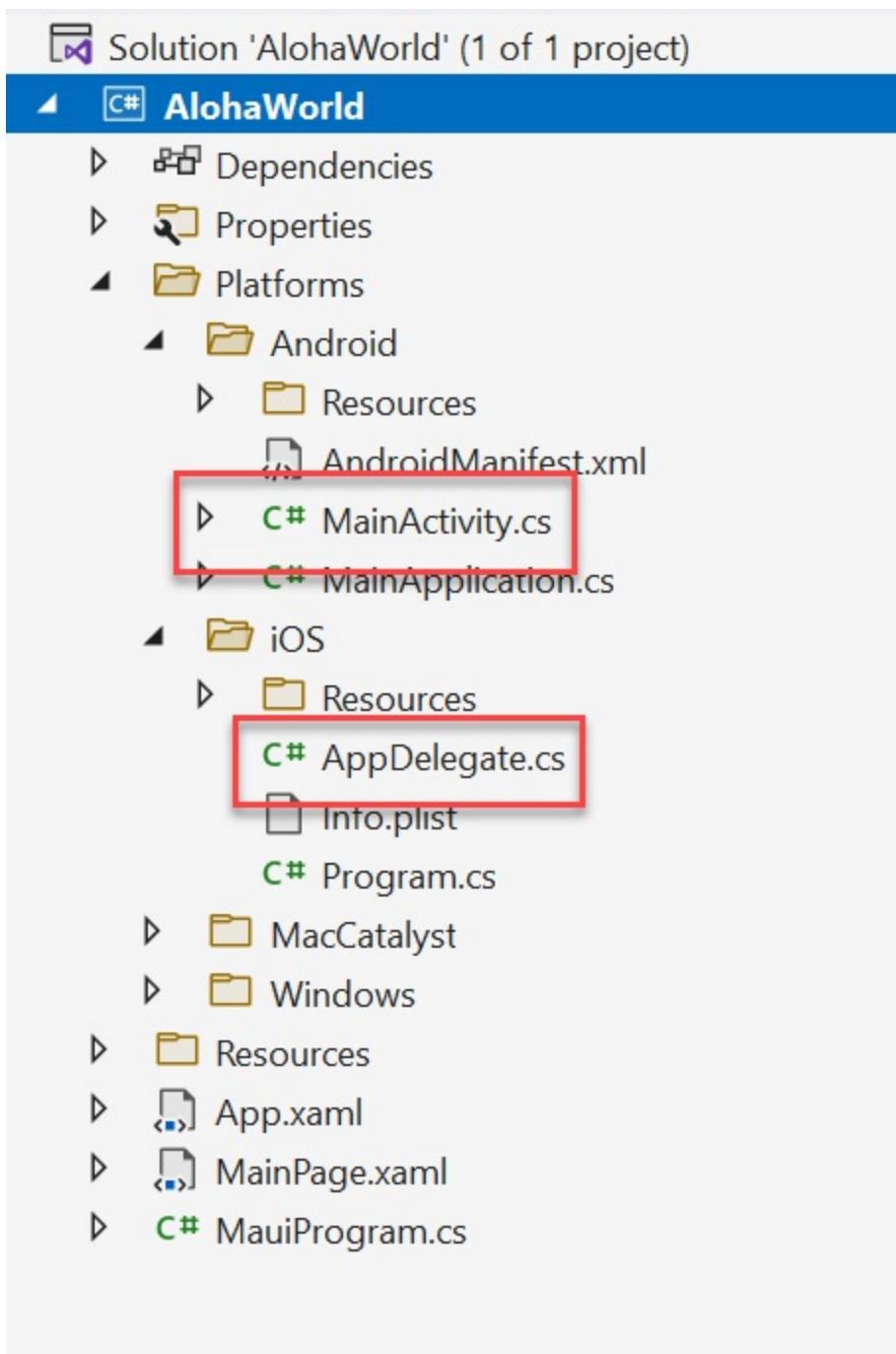


Figure 2.16 MainActivity is the entry point for Android apps and AppDelegate is the entry point for macOS and iOS apps, shown here in Solution Explorer in Visual Studio



These files are contained in a folder called Platforms, and then in a sub-folder named according to the relevant OS. .NET MAUI provides these to the OS as the entry point and loads the `MauiProgram` behind the scenes, as in Figure 2.15.

As you progress on your .NET MAUI journey you will find there are times when you need to gain a deeper understanding of these files and how they

work, and we'll dip into them throughout the book. It's important to be aware of these, but for now, and for the most part, you won't need to touch them when building .NET MAUI apps.

We understand that .NET MAUI goes from launch to getting an object of type `ContentPage` and assigning it to the `MainPage` property of the `App` class, and then displaying that page. Let's look at how `MainPage.xaml`, the `ContentPage` we assign to `MainPage` in the `App` class, is structured.

The snippet below shows the views (layouts and controls) of the `MainPage.xaml` file without any of their properties (the full code is in Listing 2.3), commented with numbers showing the corresponding to Figure 2.17.

Listing 2.3 MauiPage.xaml with all the element properties removed

```
<ContentPage>
    <ScrollView> <!-- 1 -->
        <VerticalStackLayout> <!-- 2 -->
            <Label /> <!-- 3 -->
            <Label /> <!-- 4 -->
            <Label /> <!-- 5 -->
            <Button /> <!-- 6 -->
            <Image /> <!-- 7 -->
        </VerticalStackLayout>
    </ScrollView>
</ContentPage>
```

Figure 2.17 The layouts and views used in our .NET MAU app: 1. ScrollView, 2. VerticalStackLayout, 3. Label, 4. Label, 5. Label, 6. Button, 7. Image. Size may differ depending on your screen.



Within the content page, the first child element is a `ScrollView`. A `ScrollView` is a UI container that lets the user scroll to see content that is too big to fit on screen.

```
<ContentPage>
    <ScrollView>  <!-- 1 -->
    ...

```

Inside the `ScrollView` is a `VerticalStackLayout`. A `VerticalStackLayout` is a layout component that lets you arrange views sequentially and vertically (meaning the first one appears at the top, the next one underneath it, and so on)..

```
...
    <VerticalStackLayout> <!-- 2 -->
    ...

```

Inside the `VerticalStackLayout` we have three labels, a button, and an image.

```
...
<VerticalStackLayout> <!-- 2 -->
    <Label .../> <!-- 3 -->
    <Label .../> <!-- 4 -->
    <Label .../> <!-- 5 -->
    <Button .../> <!-- 6 -->
    <Image .../> <!-- 7 -->
</VerticalStackLayout>
...
```

Listing 2.4 shows the full code of the `MainPage.xaml` file. The first thing to note is that it is an XML file. XAML (which stands for eXtensible Application Markup Language) is just XML, with some custom element names that we use for building .NET MAUI UIs.

The way that we consume those element names is by importing the relevant namespaces. This is done on lines 1 and 2 in the code.

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

On line 1 we bring in the `http://schemas.microsoft.com/dotnet/2021/maui` schema as the default namespace. This includes most of the tag names we will use. On line 2 we bring in the `http://schemas.microsoft.com/winfx/2009/xaml` schema and assign it to the `x` namespace. This includes tags for metadata about our UI, rather than the UI itself.

The first example of this can be seen on line 3, where we use the `Class` tag from the `x` namespace to associate the XAML markup file with a C# class, and you can see the corresponding ‘code-behind’ class file associated with the XAML markup file in Solution Explorer if you are using visual studio.

All views are classes

All views in .NET MAUI apps are C# classes. This includes `ContentPage` views (complete application pages) as well as controls (items displayed on a

page).

When you use a XAML file, you are using XAML markup to tell the view how to display the UI, but the view itself is still a class.

Everything you can do in XAML you can do in C#, so you can build your app without using XAML at all. That's not covered in this book, but if you like that approach you can read about it in the .NET MAUI documentation.

You can have a .NET MAUI view that is a C# file without a XAML file, but you can't have a XAML view without C#.

Lines 1-4 define the `ContentPage` tag, which is the enclosing tag for the rest of the XAML file. We've already looked at lines 1-3 and seen that they are namespace and class declarations. Line 4 sets a value to the `BackgroundColor` property of the `ContentPage` class. The value we set is a `DynamicResource` called `SecondaryColor`. The definition for this is in a `ResourceDictionary` defined in the `Resources/Styles.xaml` file; you can look at this now if you like, but we will cover styles more in Chapter 8.

Listing 2.4 MainPage.xaml

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

<ScrollView> #D
    <VerticalStackLayout Spacing="25" Padding="30">#E

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

        <Label #F
            Text="Welcome to .NET Multi-platform App UI"
            SemanticProperties.HeadingLevel="Level1"#G
            SemanticProperties.Description="Welcome to dot ne
            FontSize="18"
            HorizontalOptions="Center" />#G
```

```

<Label #H
    Text="Current count: 0"
    FontSize="18"
    FontAttributes="Bold"
    x:Name="CounterLabel"
    HorizontalOptions="Center" />

<Button #I
    Text="Click me"
    FontAttributes="Bold"
    SemanticProperties.Hint="Counts the number of times clicked"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />

<Image#J
    Source="dotnet_bot.png"
    SemanticProperties.Description="Cute dot net bot"
    WidthRequest="250"
    HeightRequest="310"
    HorizontalOptions="Center" />

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

```

We'll learn more about pages and layouts in Chapter 4, but now that you understand how a .NET MAUI app is laid out, let's make some small changes to our code and see them update in our app in real time.

Accessibility in .NET MAUI Apps

It's important to consider accessibility (a11y, as it's often shortened to) when building apps in .NET MAUI. We often rely on visual cues to provide us with meta-information about things on screen, as well as the things themselves. For example, heading size can indicate importance or hierarchy. A visually impaired user may not have access to these cues.

It's important to make our UI as visually accessible as possible. The WCAG 2.1 standard provides substantial guidance for making apps that don't just look great but are highly accessible.

Assistive technologies such as screen readers can help visually impaired

users, in some cases replacing their visual use of the screen altogether. For these users it's important to provide metadata that screen readers can interpret to replace the visual cues we described above. Semantic Properties in .NET MAUI allow us to do this. In listing 2.3 we can see how Semantic Properties are used to declare to a screen reader the heading level of a `Label`. This enables a screen reader to not just read the content of the `Label` to the user, but to also let the user know where in the hierarchy the `Label` sits, something a non-visually impaired user can discern visually.

A11y is a huge topic and one that is important to learn. While this book doesn't focus on it primarily, I will draw your attention to it whenever a relevant topic comes up.

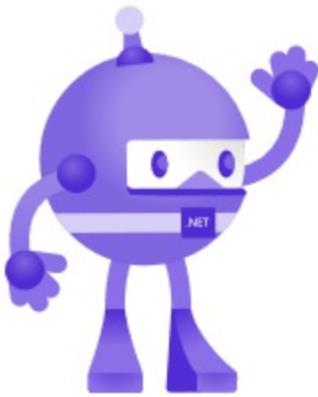
2.4 See real-time changes with hot reload

So far, we've been calling your app Aloha World, but it just says "Hello, World!". Let's change that.

The following steps will show you how to make a couple of small changes to your app while the app is running and see them reflected in real-time (without having to restart the app). As you progress as a .NET MAUI developer, this capability will become a critical part of your 'inner-development loop'. This trick is called 'Hot reload' and is currently only supported in Visual Studio and Visual Studio for Mac. The .NET CLI does not support hot reload for .NET MAUI apps.

Before we get stuck into that, one of the changes we are going to make is to the Dotnet Bot that we see on our screen (see Figure 2.18)

Figure 2.18 Dotnet Bot – the official .NET mascot



Dotnet Bot is the official mascot for .NET, and there's a cool website where you can make your own version, for use in any of your projects. Head over to <https://mod-dotnet-bot.net/> and click on the Get Started button. Build your own version of the Dotnet Bot and click on the share button to download it once you're happy.

Once you've downloaded the file, rename it to make sure it complies with .NET MAUI file naming requirements, if necessary, that the filename must contain only lower-case letters and underscores.

2.4.1 Visual Studio for Windows

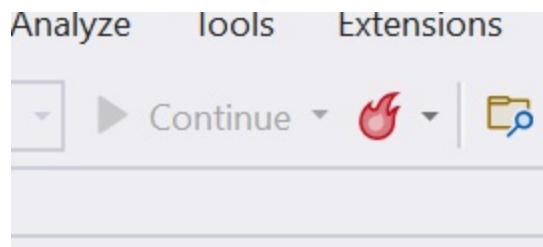
Open your app in Visual Studio if it's not open already. Choose your target platform and run your app. You should see something like we saw earlier in figure 2.10.

Now switch back to Visual Studio and follow these steps.

1. Open the file MainPage.xaml.
2. Find the Text property of the Label element that is assigned the value 'Hello, World!'. Change this to 'Aloha, World!'.
3. Switch back to your running app. You should see your label updated to show your 'Aloha, World!' message.
4. In Solution Explorer, right click on the Resources folder, and from the context menu choose Add existing item.
5. Use the file browser to select your custom Dotnet Bot that you downloaded earlier.

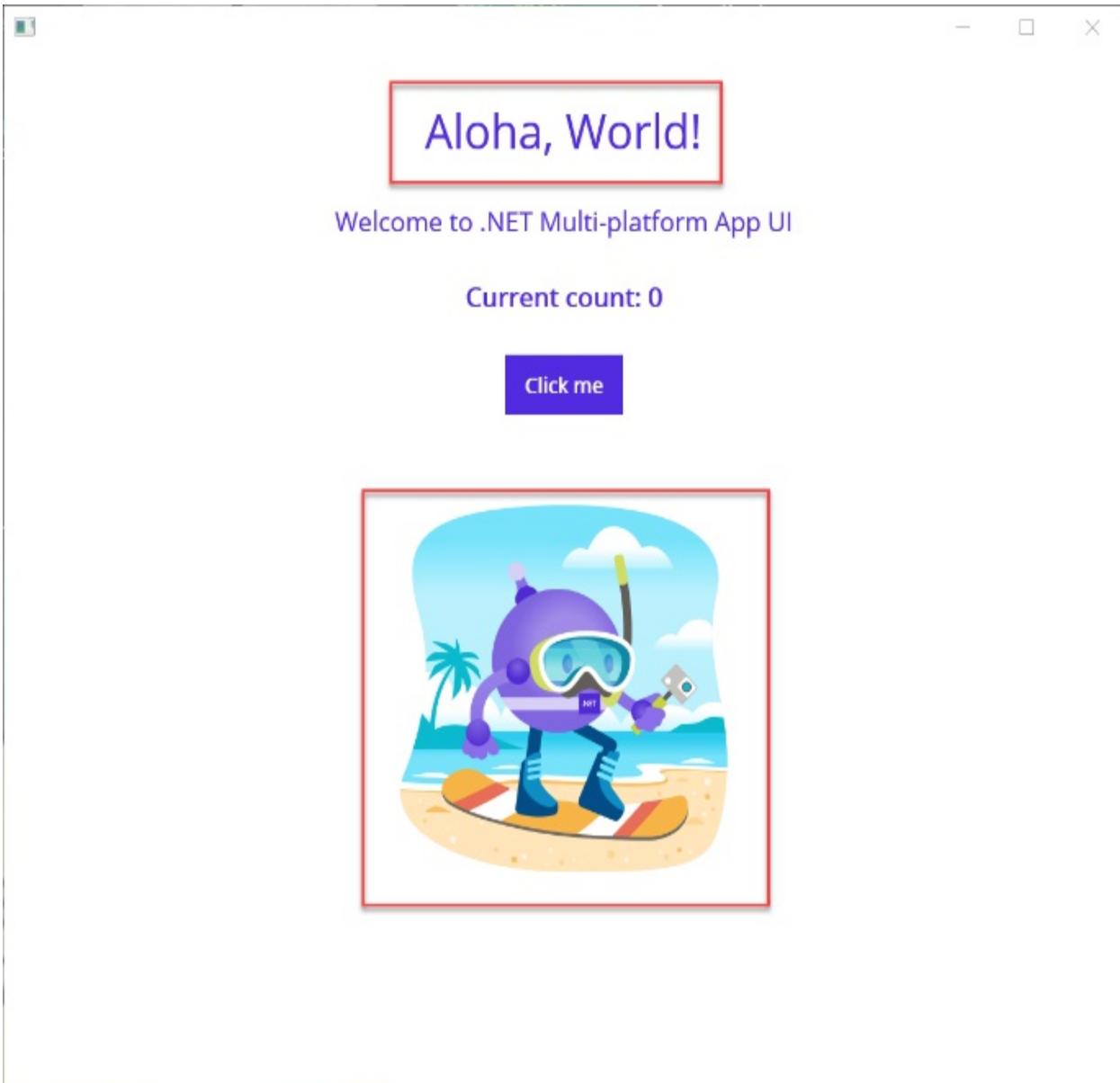
6. Once your custom Dotnet Bot is imported, right click on the image in the Resources folder and select Properties.
7. In the Properties window, ensure that the Build Action is set to MauiImage.
8. Back in the `MainPage.xaml` file, note the `Source` property of the `Image` element is set to `dotnet_bot` on line 42. Change this to the filename (including extension) of the file you imported in step 4.
9. Save the changes to the `MainPage.xaml` file.
10. To the left of the Run button on the toolbar, note a button with a red flame on it (see Figure 2.19). This is the Hot Reload button.
11. Click the Hot Reload button. At this point, Visual Studio will show you a warning, letting you know that it can't apply your changes with hot reload, and that you'll have to let it rebuild. Go ahead and rebuild, then switch to the running app to see your changes.

Figure 2.19 The Hot Reload button in Visual Studio



You should now see the changes you made reflected in your app, like in Figure 2.20.

Figure 2.20 Aloha, World!



You may have noticed that the Hot Reload button is also a dropdown. Expanding this will show you another cool option available in Visual Studio – Hot Reload on File Save. As it suggests, enabling this will automatically apply hot reload (provided that your changes are supported) when you save a file. This is the equivalent of running `dotnet watch run` for a .NET 6 (non-MAUI) app with the .NET CLI.

Why does Hot Reload need to rebuild sometimes?

Hot Reload is an awesome feature of .NET. It lets us make changes to code

while our app is running, without having to stop debugging and rebuild our app. But, as we saw above, it's not without its limitations.

When you build an application with .NET, it gets compiled to Common Language Runtime (CLR) code. This includes a kind of directory of all of your classes and their public members (methods and properties) and resources. This directory can't be altered at runtime, so while you can change the code inside a method, you can't add or remove classes or methods, or resources (such as images), without needing a rebuild.

Congratulations on making your first code changes to a .NET MAUI app. We'll go through these changes in a bit more detail throughout the book, but for now, feel free to experiment and change any other values in the XAML file you feel like. You'll quickly see how powerful the ability to change XAML layouts or code in a running app is, without having to stop and rebuild every time.

2.4.2 Visual Studio for Mac

Open your app in Visual Studio for Mac if it's not already and run the app on your Mac. You should see something similar to figure 2.10.

Now switch back to Visual Studio and follow these steps.

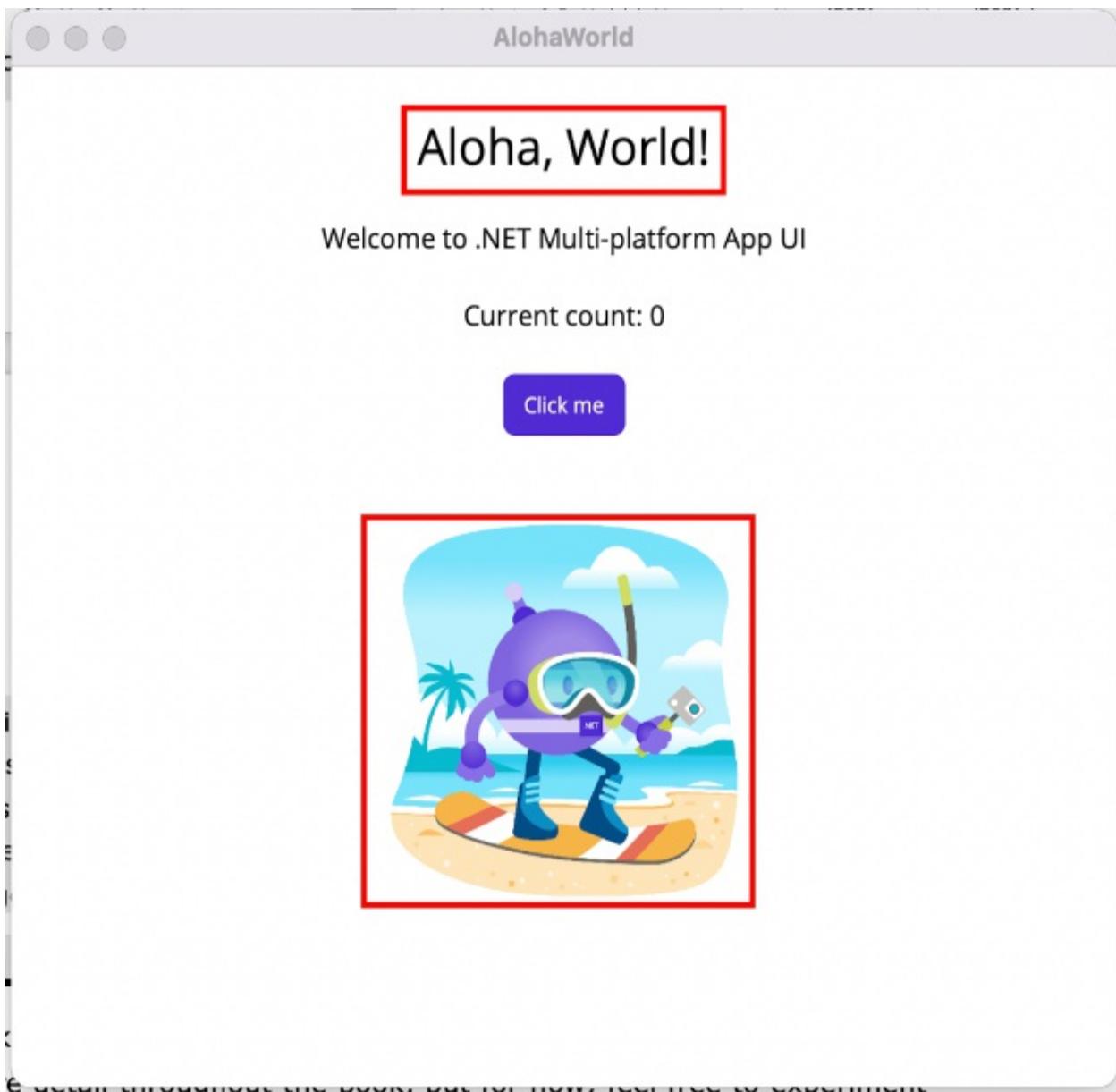
1. Open the file MainPage.xaml.
2. Find the Text property of the Label element that is assigned the value 'Hello, World!'. Change this to 'Aloha, World!'.
3. Switch back to your running app. You should see your label updated to show your 'Aloha, World!' message (see figure 2.21).
4. Back in Visual Studio, in Solution Explorer, right click on the Resources folder, and from the context menu choose Add, then Existing files....
5. Use the file browser to select your custom Dotnet Bot that you downloaded earlier.
6. Select your file, and from the Add File to Folder dialog, choose the option to copy the file to the directory.
7. Once your custom Dotnet Bot is imported, expand the Resources folder if you haven't already, and right click on the image that you added, then

select Properties.

8. In the Properties window, ensure that the Build section is expanded, then in the Build action dropdown, select MauiImage.
9. Back in the `MainPage.xaml` file, note the `Source` property of the `Image` element is set to `dotnet_bot` on line 42. Change this to the filename (including extension) of the file you imported in step 4.
10. Save the changes to the `MainPage.xaml` file.
11. Now switch back to the running Aloha, World! App. The Dotnet bot that was there before will be missing now.
12. Stop the app and start it again. This time you should see your custom Dotnet bot.

Hot reload will apply changes for you automatically while your app is running, but as you saw, changing an image didn't work. See the sidebar "Why does Hot Reload need to rebuild sometimes?" above for why this is – the same is true for macOS and Windows.

Figure 2.21 You would have seen the change from Hello, World! to Aloha, World! without having to restart the app, but you would need to restart to see the new image.



Congratulations on making your first code changes to a .NET MAUI app. We'll go through these changes in a bit more detail throughout the book, but for now, feel free to experiment and change any other values in the XAML file you feel like. You'll quickly see how powerful the ability to change XAML layouts or code in a running app is, without having to stop and rebuild every time.

2.5 Summary

- You can create a .NET MAUI app from a template provided by .NET, using the .NET CLI, Visual Studio for Mac, or Visual Studio for Windows.
- You can build and run your .NET MAUI app on any of the target platforms (iOS, Android, macOS or Windows). You can also do this from the CLI, VS for Mac, or VS for Windows.
- To choose the platform you want to run your .NET MAUI app on, use the drop-down selector in the Run button.
- You can make changes to your app while it's running. .NET Hot Reload will apply those changes in real-time without you needing to restart the app.
- You can build the UI in your .NET MAUI app using either XAML or C#. XAML is the most popular approach.
- You can leverage the host builder pattern, familiar from other .NET workloads, in .NET MAUI apps. This lets you provide initial configuration as well as using the built-in dependency injection (DI) container.

3 Making .NET MAUI apps interactive

This chapter covers:

- Defining app permissions in metadata files
- Using location, messaging and other common operating system and device features
- Saving and encrypting data to your user's device
- Connecting UI properties with Databinding
- Using collections and templates to display lists of data

In the last chapter we built and ran our first .NET MAUI app and made some small changes, but our app didn't really do anything. You may have spotted the 'Click me' button, which when clicked on increases a number displayed on screen. In this chapter we'll look at how this works and see that we can get and set values on UI elements from code.

Increasing a counter isn't the most exciting thing in the world, and it's something you can easily do with a web app. So, in this chapter we'll start to explore device capabilities that make building mobile and desktop applications with .NET MAUI fun.

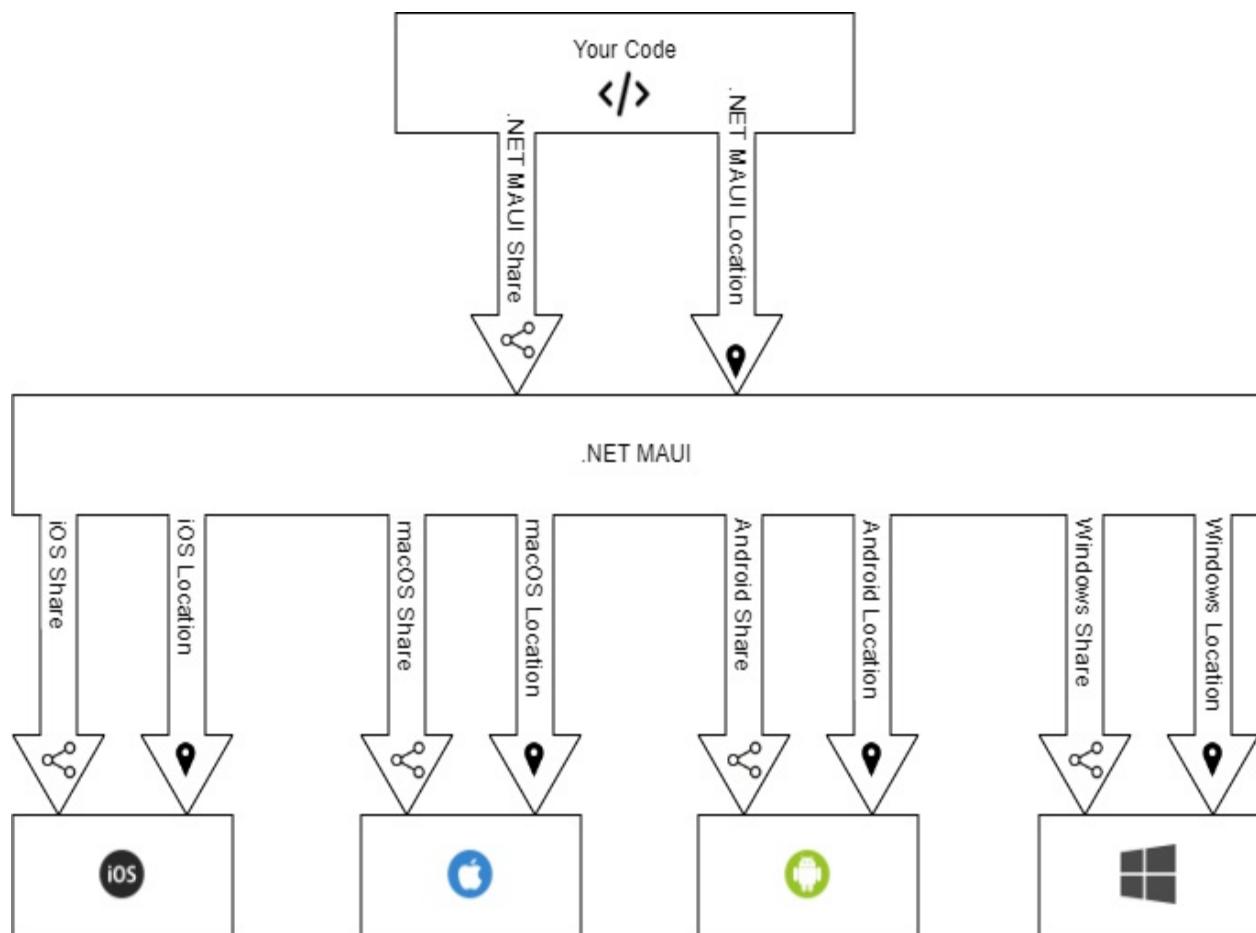
In this chapter we'll see how **databinding** is used to bind values and commands from our UI to our code. We'll start by looking at how the class corresponding to our view can easily manipulate our view, and then how this becomes more complex when introducing a ViewModel, and then we'll see how we can use databinding to overcome that.

We'll go through a couple of examples that will demonstrate this, while also learning how we can persist data locally on our device, and how we can access common device and operating system features through the .NET MAUI APIs.

3.1 Using OS and device features

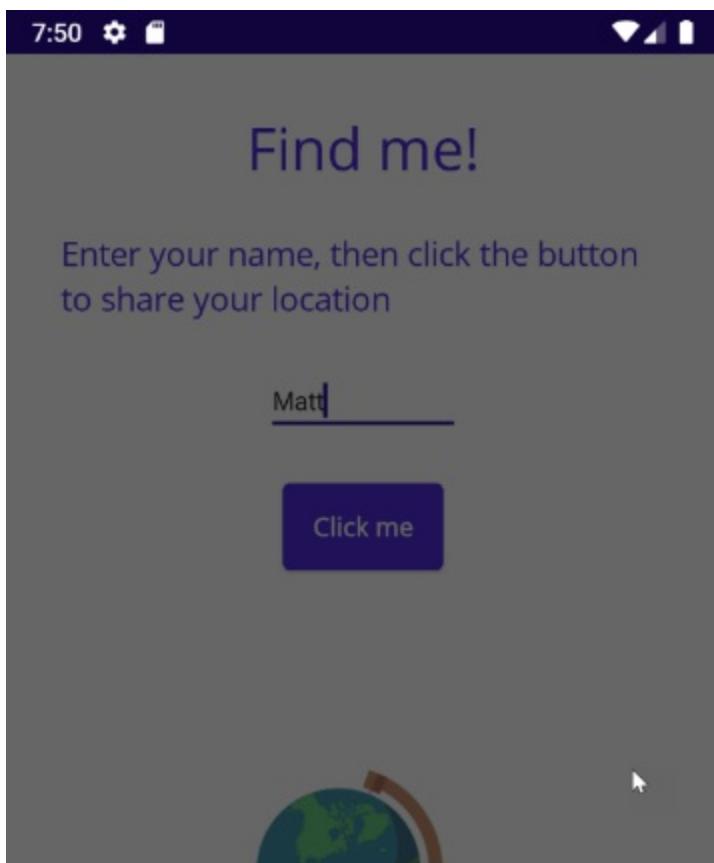
One of the reasons people choose to build a mobile or desktop app rather than a web app is for ease of access to device and operating system features. Some of these features can be accessed via web apps, and some can't; but in .NET MAUI accessing common device and OS features is simple.

Figure 3.1 .NET MAUI provides abstractions of common features (location and sharing in this example), so you can access them on all platforms using a single codebase in your app



.NET MAUI gives us access to common features found across all supported platforms. You can see an exhaustive list of capabilities provided in the Microsoft documentation, and we'll use many of these throughout this book, but in this section, we will focus on two common device features: geo-location and sharing.

Figure 3.2 The FindMe! App that lets you share your location, seen here running on Android.



Find me!



Messages



Gmail



Bluetooth



Copy to
clipboard



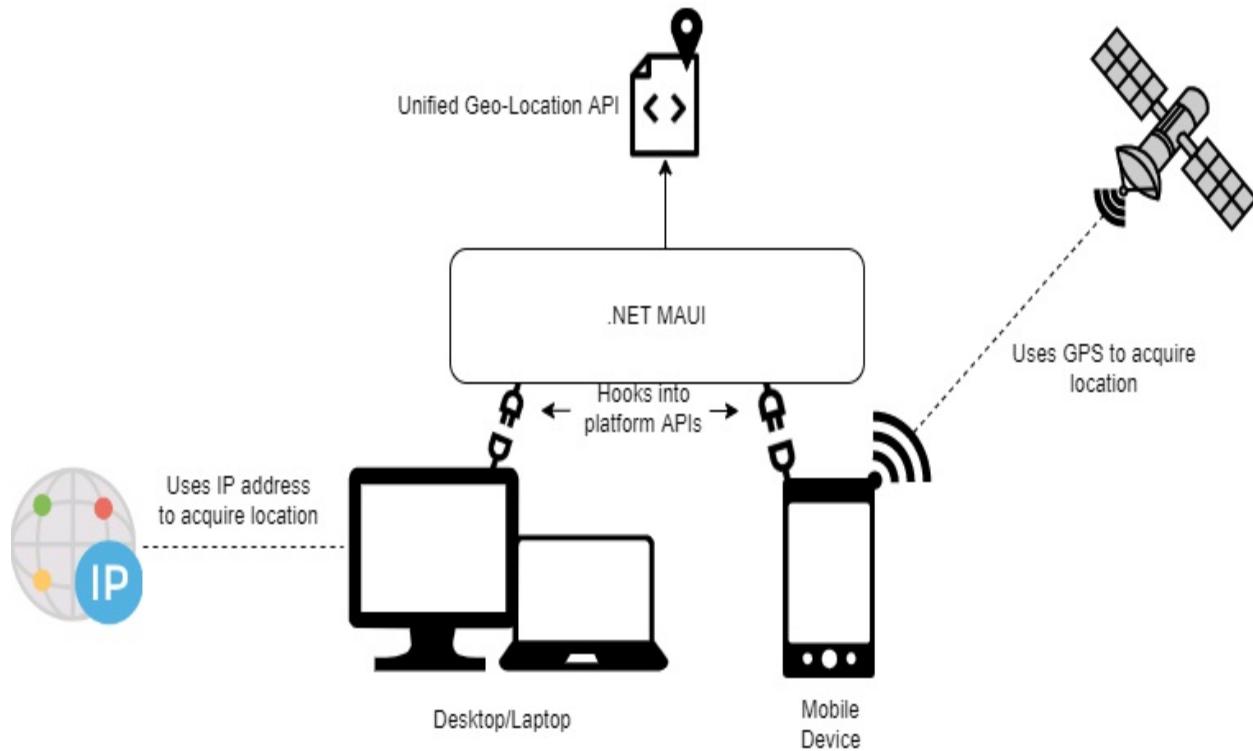
Save to Drive



To see these features in action, we're going to build an app called FindMe!,

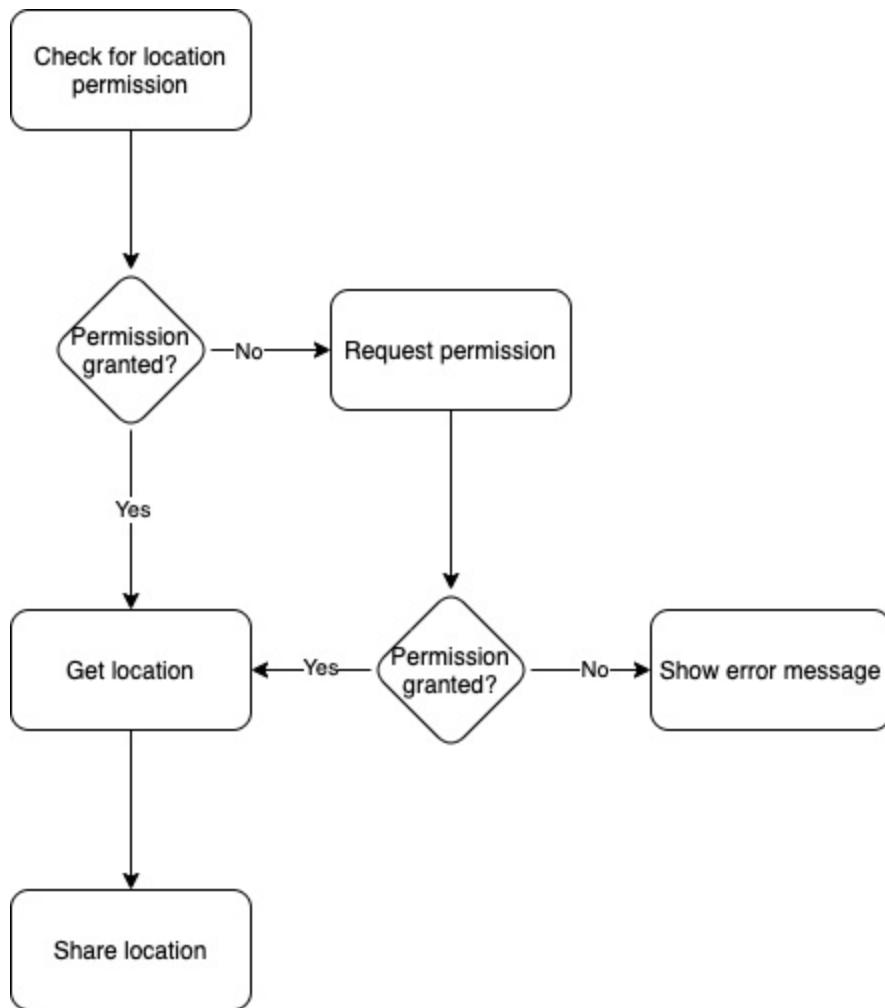
which gets the user's location and lets them share it. The user will enter their name and click a button, which will then show them multiple options for sharing their location with a friend.

Figure 3.3.NET MAUI provides a geo-location API that you call in your code. This API is an abstraction of each platform's individual API for accessing the user's location, and this in turn is an abstraction of different techniques for physically locating your user.



iOS, Windows, macOS and Android all have a geo-location API, which can be used to get information about your user's location. On a desktop your location is approximated using several data points, including your IP address and Wi-Fi information, while on a mobile device this information is combined with GPS data to provide a more precise location. With .NET MAUI, you don't need to know how these work – just that you can access them via a common abstraction.

Figure 3.4 In order to access a user's location, your app must request permission from the user. With .NET MAUI, you can check whether permission has been granted and request it from the user if not.



On all platforms, users must explicitly provide consent for your app to access their location, and on iOS and Android, your app must declare what permissions it is going to request from your users as part of its descriptive metadata. .NET MAUI provides methods to both request consent and verify that your user has consented to access their location while using the app. Declaring required permissions is done in the app's metadata rather than in code, and therefore must be done a specific way for each platform.

Figure 3.5 All supported operating systems provide a sharing API. When an app calls that API, the user is presented with a list of apps that have registered as target applications, and can choose to share the content being provided by the host application to the target application.



The other feature we will use in this chapter is sharing. All four supported operating systems provide a sharing API which, when activated, will prompt your user to select a target application. These might include email, SMS or any app that has registered a sharing capability with the operating system (social media apps, for example, nearly always expose a share capability to the OS). Having a sharing API in the OS that lets the user choose where they want to share means we don't have to cater to every sharing mechanism in every app on every OS; in .NET MAUI we can use a common abstraction that gives us access to sharing on every target platform.

Note that registering your app as a sharing target is not covered here. This is because there is no unified API provided in .NET MAUI for registering a sharing target, so you would need to write OS specific code for each platform where you wanted to do this (albeit abstracted into .NET), and the core focus of this book is on shared functionality that doesn't require OS specific code (note we do cover the basics of platform code in chapter 8, with some guidance on where to dive deeper into this topic to learn more). So for now, you'll be able to share *from* your app, but not share *to* your app.

Let's get started with building the FindMe! app. **Create a new .NET MAUI project called FindMe using the blankmaui template.** Before we start writing any UI or logic code, we need to declare what permissions we are going to request from the user. As our app is going to access the user's location, we need to declare that this is a permission that we will request from the user.

Each platform has a specific metadata file that declares information about the app that the OS and distribution platforms (e.g., the iOS App Store and the Microsoft Store) need, which, depending on the target platform, either includes a declaration of any permissions required by the app, or a declaration of which capabilities (such as network access, location, etc.) are

used by the app, from which the required permissions are inferred. Table 3.1 shows the name and location of these metadata files for each platform.

Table 3.1 The names and locations of the app metadata files for each platform

Platform	File name	Location
Android	AndroidManifest.xml	[Your app]/Platforms/Android/
iOS	Info.plist	[Your app]/Platforms/iOS/
macOS	Info.plist	[Your app]/Platforms/macOS/
Windows	package.appxmanifest	[Your app]/Platforms/Windows/

You should easily spot a pattern here – all these files are in a folder in Platforms named for each platform. This folder also contains other platform-specific files, which we'll look at in more detail later.

The app metadata files are all XML, so you can easily modify them in any text editor. However, depending on whether you are using an IDE, and if so which one, you may have access to a graphical tool that exposes visual editors for well-known values in these files. You are welcome to experiment with these, but the easiest way to make these changes is in the XML directly, and it is advisable to get comfortable with doing it this way.

Tip

If using Visual Studio, you can right-click on these manifest files and choose ‘Open with...’, then select ‘XML (text) Editor’ from the dialog that pops up.

Figure 3.6 The metadata files for .NET MAUI apps on each platform

Platform	Windows	macOS	iOS	Android
Metadata File	Package.appxmanifest	Info.plist	Info.plist	AndroidManifest.xml

Where can I learn about platform specific metadata files?

The .NET MAUI documentation includes some information about the platform specific metadata files, which is a good place to start.

As you progress as a .NET MAUI developer, you will likely encounter scenarios where you want to dig down into platform specifics, and not just for these metadata files. Throughout the book, we'll dip our toes into platform specifics here and there, which will get you started with the basics.

We'll look at this in a bit more depth in chapter 8, but it will be important to know how to find information yourself on platform specific APIs and implementation details, and how to translate these to .NET MAUI scenarios. This will be largely a self-guided process and is outside the scope of this book. But to get you started, appendix C contains a list of resources and some guidance on using them to build your .NET MAUI apps.

3.1.1 Android Metadata

Let's start with Android first – open FindMe / Platforms / Android / AndroidManifest.xml. Inside the `<manifest>...</manifest>` tags are three child elements: `uses-sdk`, `application` and `uses-permission`. The first one declares what minimum Android API version is required to run your app (for example if you want to use Android features that were only introduced in a specific version – there is also a minimum version requirement to submit apps to the Google Play store), and what the target version is.

The second element contains metadata about how your app is presented and handled by the Android OS. Here we declare where our app icon can be found (for both standard and round icons), and whether we support right-to-left (for languages like Arabic or Hebrew). And finally, whether our app should be included in a device backup (you may want to exclude it if, for example, your app contains sensitive data).

The third one is of most interest to us though. This one declares that the app will request the user's permission to access their device network state information, declaring it using the Android API specific namespace.

We're going to add some code here. First let's declare the permissions we are going to ask the user for. Listing 3.1. Shows the permissions we want to declare, with the added permissions in bold.

Listing 3.1 Permissions to add to AndroidManifest.xml

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_
<uses-permission android:name="android.permission.ACCESS_COARSE_L
<uses-permission android:name="android.permission.ACCESS_FINE_LOC
```

In addition to declaring the permissions we are going to ask for, we also need to declare the Android API features we are going to use. We'll also specify that, while the app will use these features, they will not be required in order to install the app. Add the features in Listing 3.2 after the permissions.

Listing 3.2 Features to add to AndroidManifest.xml

```
<uses-feature android:name="android.hardware.location" android:re
<uses-feature android:name="android.hardware.location.gps" androi
```

Once you have added these, your AndroidManifest.xml file should look the same as listing 3.3.

Listing 3.3 AndroidManifest.xml full code

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/andro
<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="31
<application android:allowBackup="true" android:icon="@mipmap/app
```

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_
<uses-permission android:name="android.permission.ACCESS_COARSE_L
<uses-permission android:name="android.permission.ACCESS_FINE_LOC
<uses-feature android:name="android.hardware.location" android:re
<uses-feature android:name="android.hardware.location.gps" androi
</manifest>
```

That's all the changes we need to make to the Android manifest, so you can save and close that file now. That's all the enabling changes we need for Android, now let's move on to iOS.

3.1.2 iOS Metadata

Open the **iOS** version of the `Info.plist` file. Plist is short for property list (the term used by Apple in place of manifest), and the list of properties is enclosed in a dictionary of key value pairs called `<dict>...</dict>`. The process of declaring that we need the user's location on iOS is slightly different – rather than declare both the permission and the feature, for iOS we declare the reason why we need a permission. Once we've done this, the requirements for both the feature and the permission are implicit. Add the key value pair from listing 3.5 to the end of the dictionary in this `Info.plist` file.

Listing 3.5 Key value pair to add to the iOS Info.plist file

```
<key>NSLocationWhenInUseUsageDescription</key>#A
<string>We need your location in order to share it.</string> #B
```

Close and save the file, and then make the same change to the **MacCatalyst** version of the `Info.plist` file.

Why is the Mac platform called MacCatalyst?

A significant portion of Apple's revenue comes from app sales, most of which comes from the iOS App Store. In an effort to increase the variety and volume of apps available on the Mac App Store, Apple introduced the Catalyst program, which enables iOS developers to package their apps for the Mac. In fact, starting in 2021, all apps submitted to the iOS App Store are marked for release on the Mac App Store too by default – developers must

opt their app out of this option (more on this in chapter 8). But many developers appreciate the convenience.

.NET MAUI takes advantage of this and uses the Catalyst platform to package apps for the Mac. The outcome for developers and users is the same as if the app had been written specifically for the Mac, but under the hood it means that .NET MAUI can provide an abstraction of the UI SDK used in iOS (`UIKit`), rather than having to also provide an abstraction of the macOS UI SDK as well (`AppKit`). The situation may change in the future and Apple may unify these SDKs, but in the meantime, Catalyst provides a convenient way for .NET MAUI to package your app for the Mac.

3.1.3 Windows Metadata

The last platform we need to specify permissions for is Windows. In the Windows platform folder, open `Package.appxmanifest`.

`Package.appxmanifest` is another XML file (the same as the other platforms). Toward the bottom of the file, you'll see a `Capabilities` node. Inside this node, under any existing capabilities, add the following line:

```
<DeviceCapability Name="location" />
```

That's all that is needed to tell Windows apps that we will ask for the user's permission. All our platform setup is now complete, so let's move on to something a little more interesting. We'll start by making some small changes to the UI, and then go on to writing the code that gets the user's location and calls the sharing API.

3.2 The FindMe UI

Open the `MainPage.xaml` file. This contains the UI definition for the page, and we need to make a few small changes to adapt this UI to suit our FindMe app. First, let's change the title displayed on our page. The very first element in the `VerticalStackLayout` is a `Label`, with its `text` property set to "Hello, world!". Change this text to "Find me!". The next `Label` is a kind of sub-title (although it is semantically set to the same title level), with text that reads "Welcome to the dot net Multi Platform App UI". Change this text to "Enter

your name, then click the button to share your location”. Note that the text is also entered into a property called `SemanticProperties.Description`; change the value here as well to match what you entered in the `Text` property.

Semantic Properties

Semantic properties are descriptive values that are used by assistive technologies such as screen readers to help people use your app. For example, someone who is vision impaired may not be able to easily distinguish color, so typical conventions, such as a red label to indicate an error, or a green tick to indicate success, would not work for some users. Color, and in particular contrast, can be used to enhance the readability of your app, but you can't rely on any one technique alone. Accessibility (or a11y as it's often shortened to) is a big topic that you will need to invest some time in. We talked about these a little more in chapter 2, but Microsoft provides a variety of resources in this area, including documentation and videos, and much of it is specific to .NET MAUI. A great place to start is the Accessibility page in the Fundamentals section of the official .NET MAUI documentation. Spending some time to get familiar with this is essential.

The next UI element is a label that shows a counter value. Replace this `Label` control with the following snippet:

```
<Entry#A  
    Placeholder="Enter your name"#B  
    SemanticProperties.Hint="Enter your name to be used when shar  
    HorizontalOptions="Center"  
    x:Name="UsernameEntry"/>#C
```

.NET MAUI Naming Conventions

Microsoft provide some coding convention guidelines for each of their .NET languages, and there are some XAML syntax conventions available for UWP, but not for .NET MAUI (or even Xamarin.Forms). With that said, there are some best practices that have become the de facto standard across the industry. You should become familiar with these, but ultimately you and/or your team will need to define what is the best approach for your product.

In this book, we will use PascalCase for all public properties, underscored

`_camelCase` for private fields, and for XAML properties, will use PascalCase with the convention of [property or action][ElementType]. For example, `LoginButton` or `UsernameEntry`.

Following these conventions will make it easier for anyone reviewing your code, including yourself, to understand what a property is for and what it does.

Change the value of the `Clicked` property on the `Button` element. It currently refers to a method in the code behind file called `onCounterClicked`; change this to `OnFindMeClicked`. This method doesn't exist yet but we're going to create it shortly. You will also need to update the semantic hint; change the `SemanticProperties.Hint` value to "Presents apps available to share your name and location via."

Finally, for a bit of flair, let's replace the dotnet bot with an image more fitting to our scenario. In chapter 2, we looked at customizing this bot, you can use a custom bot here if you wish, or use any image of your choosing (that you have the right to use, of course!). For example, you may want to use something that resembles the commonly used location pin icon, or a globe (as used in the screenshots in this example – if you want to use this actual image, you can find it in the book's online resources). Add your image to the `Resources/03images` folder, the same way we did in chapter 2, and update the `Source` property on the `Image` element in the UI grid to the filename of your new image.

Now let's make some changes to the code behind file. Open `MainPage.xaml.cs`. When we share our location, we're going to send someone a link that will open our location in Bing Maps. Bing provides a URL format that supports opening a location with latitude and longitude coordinates, so let's add a field to hold the base URL. Add this at the top of the `MainPage` class.

```
string baseUrl = "https://bing.com/maps/default.aspx?cp=";
```

We'll also need a variable to hold the user's name, that they provide in the `Entry` we added to the XAML. Add this property after the `baseUrl` field.

```
public string UserName { get; set; }
```

Now let's add a method that does the actual work of getting the user's name and sharing their location, as shown in Figure 3.7. Add the method from listing 3.6 to the end of the MainPage class.

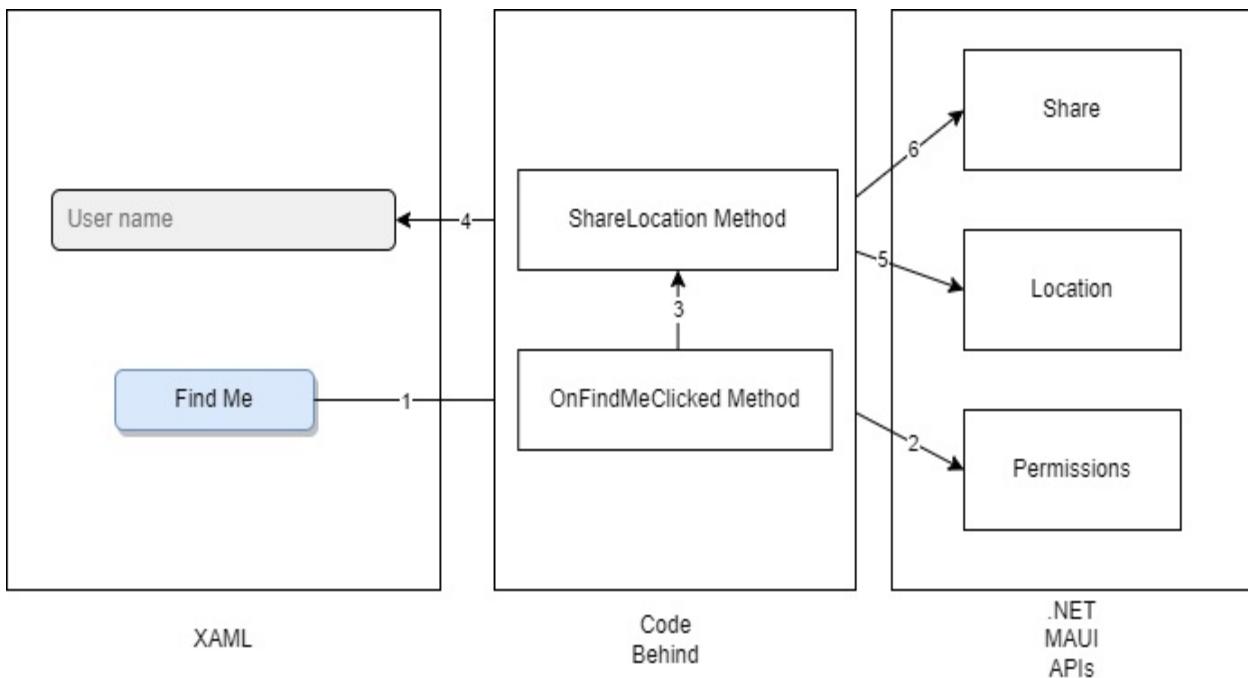
Listing 3.6 ShareLocation method

```
private async Task ShareLocation()
{
    UserName = UsernameEntry.Text;#A

    var locationRequest = new GeolocationRequest(GeolocationAccuracy.#B
    var location = await Geolocation.GetLocationAsync(locationRequest

    await Share.RequestAsync(new ShareTextRequest#D
    {
        Subject = "Find me!",#E
        Title = "Find me!",#E
        Text = $"{UserName} is sharing their location with you",#F
        Uri = $"{baseUrl}{location.Latitude}~{location.Longitude}"#G
    });
}
```

Figure 3.7 When the user clicks the Find Me button, an event is raised and the OnFindMeClicked event handler in the code behind is triggered (1). This uses the .NET MAUI permissions APIs to verify that the user has granted the app permission to access location (2). It then calls the ShareLocation method (3), which gets the user's name from the UsernameEntry field in the UI (4), and the location from the Location API (5) and shares the user's name and location using the Share API (6).



Now that we've created the method to share the user's location, we need a way to call it. We've already told the UI to call a method called `OnFindMeClicked` in the XAML, so let's add it and use this to call the `ShareLocation` method. We can get rid of the `OnCounterClicked` method, as we are not using it anymore, and replace it with the `OnFindMeClicked` method shown in Listing 3.7. You can also get rid of the `count` field, which we are not using anymore either.

Listing 3.7 OnFindMeClicked method

```

private async void OnFindMeClicked(object sender, EventArgs e)#A
{
    var permissions = await Permissions.CheckStatusAsync<Permissions.

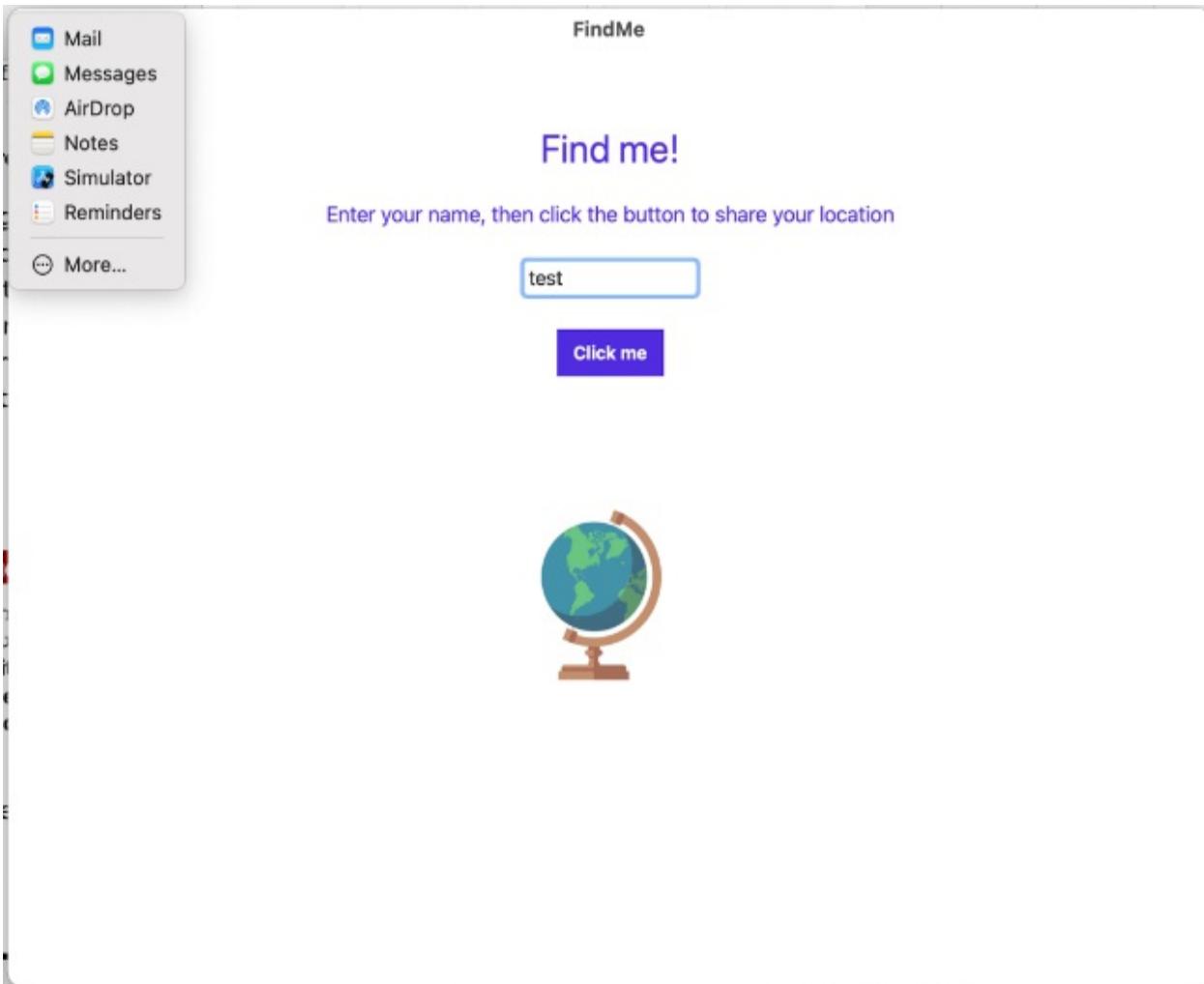
    if (permissions == PermissionStatus.Granted)#C
    {
        await ShareLocation();#D
    }
    else
    {
        await App.Current.MainPage.DisplayAlert( "Permissions Error", "Yo
        var requested = await Permissions.RequestAsync<Permissions.Locat
        if (requested == PermissionStatus.Granted)#G
    }
}

```

```
{  
await ShareLocation();#H  
}  
else  
{  
  
if (DeviceInfo.Platform == DevicePlatform.iOS || DeviceInfo.Platf  
{  
await App.Current.MainPage.DisplayAlert("Location Required", "Lo  
}  
else  
{  
await App.Current.MainPage.DisplayAlert("Location Required", "Lo  
}  
}  
}  
}  
}
```

We have now finished writing all the code we need for the FindMe app. Congratulations! Run your app and have fun sharing your location.

Figure 3.8 The finished FindMe app running on macOS



3.3 Persisting data on your user's device

At the start of the last section, we talked about how we can access device and OS features not available to web apps. While this is true, the example we used - accessing a user's location - is in fact something you can do in a web app. In this section we're going to explore storing data. This is something you can also do in a web app, but there are limitations that we don't have in .NET MAUI.

Table 3.2 a comparison of data storage options in web apps and .NET MAUI apps

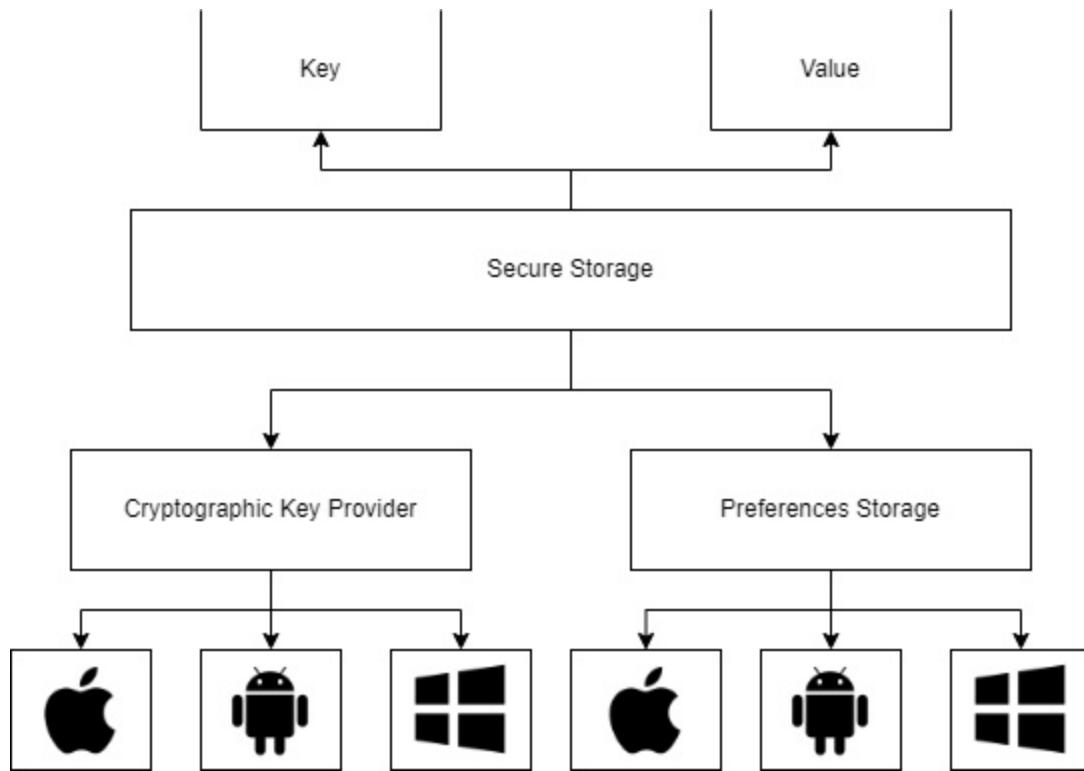
Storage			
---------	--	--	--

Feature	Web App	.NET MAUI App
Files	A user can open and save files. File system access is inconsistent between browsers and operating systems, so access to specific files cannot be guaranteed. The app cannot open a file without the user.	A user can open and save files. Abstractions are provided to common locations, meaning a developer can expect a consistent result across platforms. The app can open a file without a user, so data necessary for the app's operation can be loaded in the background.
Preferences	Can be stored in the browser cache, would need to be serialized to a structured text format like json. Can also use the Web Storage API.	A common abstraction is provided to the app configuration location on each platform. Data is stored in key value pairs.
Encryption	Web apps cannot store encrypted data securely. There are ways to encrypt data in a web app, but if you want your app to also be able to read that data, the app needs to store the key, which would be exposed.	A common abstraction is provided that lets you store data in an encrypted location, with a key that is managed by the OS.

Structured Data	Web apps can use the IndexedDB API, an object-oriented database API available in modern web browsers.	The .NET ecosystem makes countless options available, including EF Core. Any database engine available in the .NET Standard can be used in a .NET MAUI app.
-----------------	---	---

In table 3.2 we can see that the major distinguishing feature of a .NET MAUI app, when compared to a web app, is secure storage. Web apps are constrained by the permissions and APIs available to web browsers and cannot store encrypted data in a way that is both secure and reversible. If a web app wants to access encrypted data, it must store the key in the browser's cache, which is susceptible to a number of attacks. Platform executable apps, on the other hand, have full access to all the APIs provided by the OS or platform, and are only constrained by the permissions granted to them by the user.

Figure 3.9 SecureStorage uses a cryptographic key provided and managed by the operating system to encrypt the value of an entry stored using the Preferences API.



.NET MAUI provides a feature for this called `SecureStorage`. `SecureStorage` uses key value pairs, just like `Preferences` but, unlike `Preferences`, a cryptographic key is obtained from the OS to encrypt the value of the stored data. `SecureStorage` uses an abstraction of each platform's underlying encrypted data management API. On macOS and iOS, for example, `SecureStorage` uses the KeyChain, while on Android it uses Keystore. The cryptographic key is managed by the operating system and is only available to the app, and in most cases is backed by a hardware encryption chip.

Encryption with .NET MAUI apps

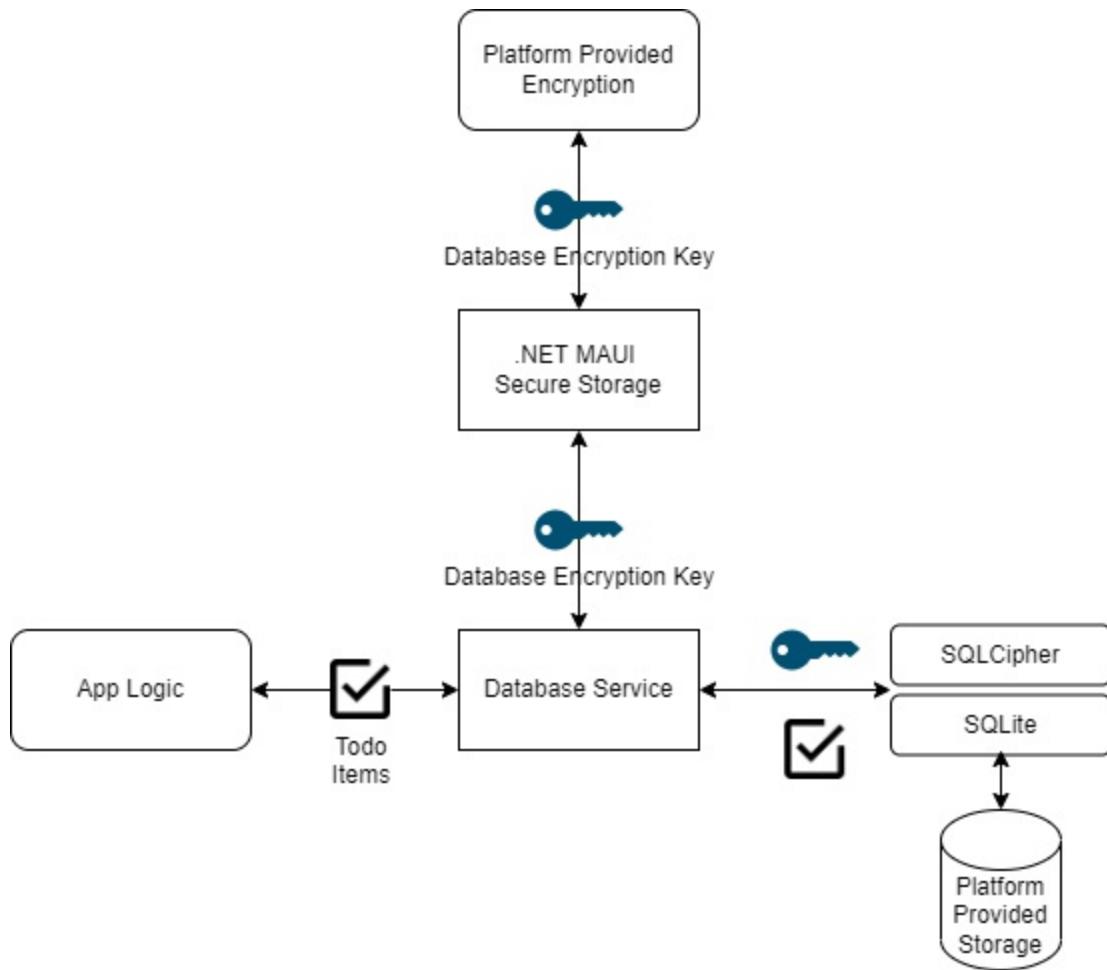
On iOS and macOS, hardware-based encryption is guaranteed as all macOS and iOS devices ship with a built-in encryption chip. On Android, especially if the device is a tablet or phone, hardware encryption is highly likely, but not guaranteed. Most modern Windows laptops have a Trusted Platform Module (TPM) chip that provides hardware encryption, while relatively few desktops do. Windows 11's TPM requirement will likely shift this balance in the future.

This is useful information, but not strictly necessary for building apps with .NET MAUI. .NET MAUI provides `SecureStorage` as an abstraction, so whether hardware or software based, the API you use in your code is the same.

While encryption is the clear standout differentiator between web apps and native apps like those build with .NET MAUI, structured data storage is also more versatile in .NET MAUI apps. The `IndexedDB` provided in modern web browsers is powerful, but it is still a collection of key value pairs, albeit an indexed collection which provides fast search and retrieval. The range of database options available in .NET MAUI apps is significantly broader and more varied, offering object-relational mappers (ORMs) and relational databases, as well as plenty of NoSQL options.

In the remainder of this section, we are going to build `MauiTodo`, a simple to-do app for creating a list of tasks. We'll use a database to store our to-do items so that they are persisted between uses, and because we want to include confidential to-do items, we will encrypt the database and store our to-do items securely. Figure 3.10 shows the high level architectural design for the `MauiTodo` app.

Figure 3.10 The `MauiTodo` app will use SQLite to store data. The data will be encrypted using SQLCipher. The database encryption key will be a GUID, which itself will be encrypted and stored by the OS and accessed with the .NET MAUI `SecureStorage` API.



In MauiTodo, the app that we’re going to build in this section, we will use **SQLite**, a popular open-source database engine. You could use the EF core SQLite provider, but we’ll use a package called **SQLite-net**, an implementation of SQLite specifically designed for mobile apps. We’ll also add **SQLCipher** to encrypt the contents of our database. SQLCipher will handle the encryption and decryption of the data for us, but we’ll use the **SecureStorage API** to let the OS manage the database encryption key for us.

Let’s start by creating a new .NET MAUI project called MauiTodo using the `blankmaui` template. Next we’ll define the data model. We’ll want a class to represent our to-do items, so **add a folder called `Models`, and add a new class in this folder called `TodoItem.cs`**. Listing 3.8 shows the code for this class, so update your class to look like this.

Listing 3.8 The TodoItem class

```
using System;

namespace MauiTodo.Models
{
    public class TodoItem
    {
        public int Id { get; set; }

        public string Title { get; set; }

        public DateTime Due { get; set; }

        public bool Done { get; set; } = false;
    }
}
```

We don't need to go through this in detail – you can see that it's a plain-old CLR object (POCO) with some properties that represent a typical to-do item. Now that we've got the model, we want to add the database functionality to store and retrieve our to-do items. As mentioned above, we're going to use SQLite, so **add the sqlite-net-pcl and sqlite-net-sqlcipher NuGet packages.**

Consuming nuget packages

One of the things that makes .NET MAUI an awesome tool for .NET developers is having access to the .NET ecosystem. A significant component of this is the plethora of libraries available as NuGet packages.

Any NuGet package that you are familiar with and like using in your code is available for you to use in your .NET MAUI apps, provided it doesn't have a dependency on a specific platform. One example might be the Windows Compatibility Pack, and more broadly, some packages popular with Xamarin.Forms developers have a Xamarin.Forms dependency. These won't work with your .NET MAUI apps.

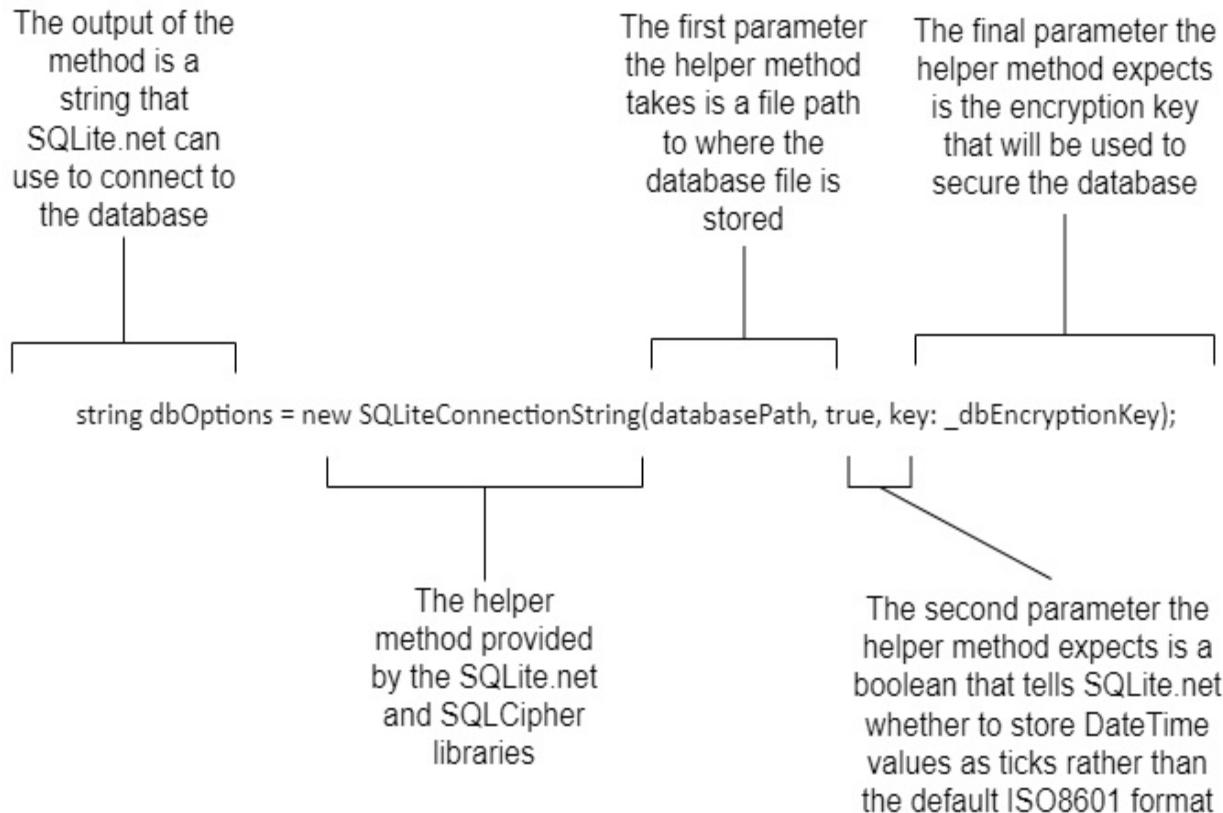
If you are coming from Xamarin.Forms, one major advantage with .NET MAUI is the single project solution. If you want to consume a NuGet package, you need import it only once, rather than once for your shared project and once for each platform.

Now **create a folder called** Data, and in this folder **create a new class called** Database.cs. In this class, we're going to add methods for adding, retrieving and deleting to-do items from a SQLite database, with a data file that's stored in a special location for storing user data. The location is different on each platform, but we'll use an abstraction in .NET MAUI to set the path automatically. We'll also initialize the database with an encryption key, which will be a GUID, and we'll encrypt and securely store the GUID with SecureStorage.

SQLite needs a connection string to connect to a database. In our case this will include the path to the data file which we are working directly with and will also define how DateTime values will be stored. As we are using SQLCipher, we will also provide the key used to encrypt and decrypt the database.

SQLite-net includes helper methods that generate this connection string for us based on values we provide, and SQLCipher has additional extension methods covering the encryption key.

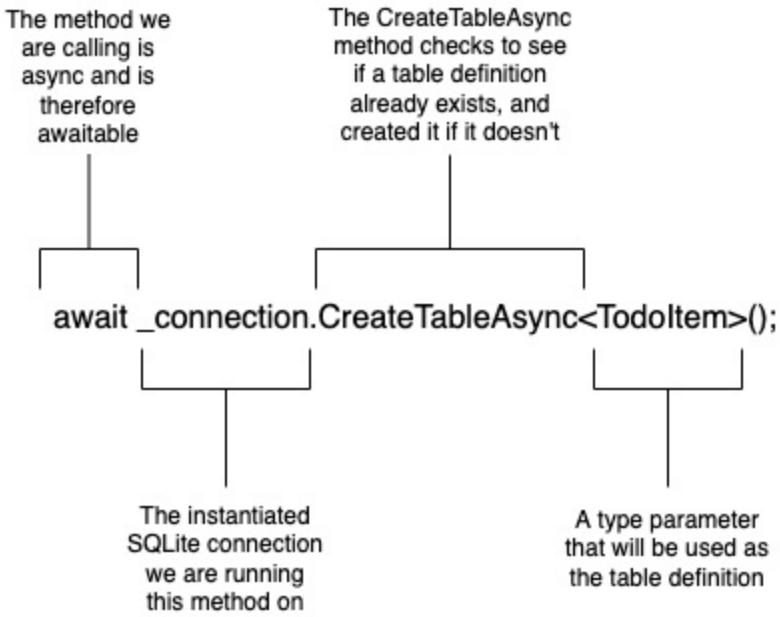
Figure 3.11 SQLite-net provides helper methods for constructing connection strings. SQLCipher provides additional extensions that enable the construction of connection strings that include encryption keys.



SQLite includes a lightweight ORM, which allows us to interact with the database using POCOs rather than having to write SQL statements every time we want to write data to or retrieve data from the database, and it also means we can use those POCOs as our table definitions.

We've defined a model in our to-do app using a C# type that represents our to-do items, and we'll let SQLite generate tables for us using this definition. We want to be confident that our table has been created in the database before we start reading from or writing to it.

Figure 3.12 SQLite features an ORM that lets us define tables using POCOs. An asynchronous method is provided to verify that a table exists and create it if it doesn't.



While we need our table before we can start using it, we don't need to block our app from continuing to run while the table is being set up. Fortunately, the method provided by SQLite-net for creating the table is asynchronous, but as we can't have async constructors, we'll wrap the call that creates the table in its own async method. We can then call this method from our constructor using a discard (_), which will then run the method in a thread without blocking the UI.

Async/await in .NET MAUI apps

As a UI developer, it's important to ensure that long running (or possibly failing) operations don't lock up the app. This is a poor user experience, and you will lose users. As a .NET developer you may already be familiar with `async/await` and asynchronous programming.

In either case, it's well worth spending some time learning how these patterns are best applied in UI development. Throughout the book we'll see some more examples, and I'll call out specific patterns, tips, or traps as we encounter them. If you want to learn more, there are plenty of excellent resources available online (especially from Brandon Minnick and Brian Lagunas). You can find these by searching for their names along with `async` or `await` as keywords. Another excellent resource is SSW Rules, a free resource provided by my employer, which you can also find easily by

searching.

Links to all of these are available in the book's online resources.

Once our table is created, we can continue to use POCOs and the ORM to interact with it. Adding items is as simple as calling `_connection.InsertAsync(item)`; the ORM is smart enough to infer the table we want to insert `item` into based on its type. Querying a table for entries is simple too, using a LINQ query and a lambda expression (see “LINQ and lambda expressions” below).

The full code for the `Database.cs` file is shown in listing 3.9.

Listing 3.9 the Database class

```
using MauiTodo.Models;
using SQLite;

namespace MauiTodo.Data
{
    public class Database
    {
        private readonly SQLiteAsyncConnection _connection;

        public Database()
        {
            var dataDir = FileSystem.AppDataDirectory;#A
            var databasePath = Path.Combine(dataDir, "MauiTodo.db");

            string _dbEncryptionKey = SecureStorage.GetAsync("dbK

            if (string.IsNullOrEmpty(_dbEncryptionKey))
            {
                Guid g = new Guid();#C
                _dbEncryptionKey = g.ToString();
                SecureStorage.SetAsync("dbKey", _dbEncryptionKey)
            }

            var dbOptions = new SQLiteConnectionString(databasePa
            _connection = new SQLiteAsyncConnection(dbOptions);#F

            _ = Initialise();#G
        }
    }
}
```

```

private async Task Initialise()
{
    await _connection.CreateTableAsync<TodoItem>();#H
}

public async Task<List<TodoItem>> GetTodos()
{
    return await _connection.Table<TodoItem>().ToListAsync()
}

public async Task<TodoItem> GetTodo(int id)
{
    var query = _connection.Table<TodoItem>().Where(t =>
        return await query.FirstOrDefaultAsync();#K
    }

    public async Task<int> AddTodo(TodoItem item)
    {
        return await _connection.InsertAsync(item);#L
    }

    public async Task<int> DeleteTodo(TodoItem item)
    {
        return await _connection.DeleteAsync(item);
    }

    public async Task<int> UpdateTodo(TodoItem item)
    {
        return await _connection.UpdateAsync(item);
    }
}
}

```

We've now got a database in our app, that can securely store our secret to-do items, and provides methods that can be used to store and retrieve to-do items. The next step is to build a UI to display and enter to-do items, and some code to connect that UI to the database

LINQ and lambda expressions

As .NET MAUI apps are .NET apps, we can use everything in the C# toolbox. If you've worked with EF Core you are no doubt familiar with

language integrated query (LINQ). If not, it's well worth investing some time in learning this (you can start with the Microsoft documentation), as it's useful in several scenarios. These scenarios are not just related to databases; LINQ also works with collections, and collections are at the heart of nearly all apps.

Lambda expressions let you declare an anonymous function. This means that you can tell your code to go and execute some other code and return the result, without having to declare that other code as a method. Lambda expressions are incredibly powerful when used with LINQ, but are also useful for a range of scenarios. I would also encourage you to read the Microsoft documentation on lambda expressions if they are not something you are familiar with.

As you can see, any .NET skills are transferrable to .NET MAUI apps. If you're a .NET developer, then you're a .NET MAUI developer!

In the `MainPage.xaml` file, we're going to make some changes to the UI. We'll update the page title, add a text entry where the user can specify the title for new to-do items, and a date picker for the due date. We'll need a button to confirm adding new to-do items, and finally we'll want a way to display a list of the to-do items stored in the database.

We're going to add these ourselves, rather than modifying the existing controls. **In the XAML, delete the ScrollView and everything inside it.** This should leave you with the `<ContentPage...>...</ContentPage>` tags. The `ScrollView` that we removed is a layout, and we're going to replace it with a different kind of layout called a `Grid`. We'll learn more about layouts in chapter 4, but for now, **add an opening and closing Grid tag with the properties and child elements shown in listing 3.10.**

Listing 3.10 the UI for the MauiTodo app

```
<Grid RowDefinitions="1*, 1*, 1*, 1*, 8*" #A  
      MaximumWidthRequest="400"  
      Padding="20">  
  
    <Label Grid.Row="0" #B  
          Text="Maui Todo"
```

```

        SemanticProperties.HeadingLevel="Level1"
        SemanticProperties.Description="Maui Todo"
            HorizontalTextAlignment="Center"
            FontSize="Title"/>

    <Entry Grid.Row="1">#C
        HorizontalOptions="Center"
        Placeholder="Enter a title"
        SemanticProperties.Hint="Title of the new todo item"
        WidthRequest="300"
        x:Name="TodoTitleEntry" />

    <DatePicker Grid.Row="2">#D
        WidthRequest="300"
        HorizontalOptions="Center"
        SemanticProperties.Hint="Date the todo item is due"
        x:Name="DueDatepicker" />

    <Button Grid.Row="3">#E
        Text="Add"
        SemanticProperties.Hint="Adds the todo item to the database"
        WidthRequest="100"
        HeightRequest="50"
        HorizontalOptions="Center"
        Clicked="Button_Clicked"/>

    <ScrollView Grid.Row="4">#F
        <Label HorizontalTextAlignment="Center"
        SemanticProperties.Description="The list of todo items in
            x:Name="TodosLabel" />#G
    </ScrollView>

</Grid>

```

We've now defined a simple UI for entering and listing the user's to-do items. All the controls in this UI are familiar, except the `DatePicker` which is new. A `DatePicker` allows the user to choose a date with a graphical selector rather than entering it into a text field.

Now that the UI is defined, lets add some code to the code behind to wire up some functionality.

- We'll start by adding some properties and fields. These will hold an instance of the database, as well as the values of to-dos in the database, and any values the user wants to add to a new to-do item.

- We'll update the class constructor in the code behind file as well as adding a method to initialize our page on load.
- Finally, we'll add a method to respond to button clicks and add a new to-do to the database. Update your code behind file (`MainPage.xaml.cs`) to match listing 3.11.

Listing 3.11 the code for the MauiTodo main page

```
using MauiTodo.Data;
using MauiTodo.Models;

namespace MauiTodo
{
    public partial class MainPage : ContentPage
    {
        string _todoListData = string.Empty;#A

        readonly Database _database;#B

        public MainPage()
        {
            InitializeComponent();
            _database = new Database();#C

            _ = Initialise();#D
        }

        private async Task Initialise()
        {
            var todos = await _database.GetTodos(); #E

            foreach (var todo in todos)
            {
                _todoListData += $"{todo.Title} - {todo.Due:f}{En
            }
        }

        TodosLabel.Text = _todoListData;#G
    }

    private async void Button_Clicked(object sender, EventArgs
    {
        var todo = new TodoItem#H
        {
            Due = DueDatepicker.Date,
            Title = TodoTitleEntry.Text
        }
    }
}
```

```
    };

    var inserted = await _database.AddTodo(todo);#I

    if (inserted != 0)#J
    {
        _todoListData += $"{todo.Title} - {todo.Due:f}{En

        TodosLabel.Text = _todoListData;#L

        TodoTitleEntry.Text = String.Empty;#M
        DueDatepicker.Date = DateTime.Now;#M
    }
}

}

}
```

This completes our MauiTodo app for now. Run the app and you should see something that looks like figure 3.13. You can add multiple to-do items, give them all different due dates, and see them appear in the list at the bottom of the screen.

Figure 3.13 MauiToDo running on Windows. When you click on the Add button, an event handler gets the value from the values from the Entry and Datepicker and uses them to create a new to-do item, with those values for the title and due date respectively. It then adds them to the list of to-do items on the screen.

Maui Todo

Enter a title

18/12/2021



Add

Write - Thursday, 7 April 2022 12:00 AM

Finish .NET MAUI in Action - Saturday, 1 January 2022

12:00 AM

3.4 Databinding – connecting UI to code

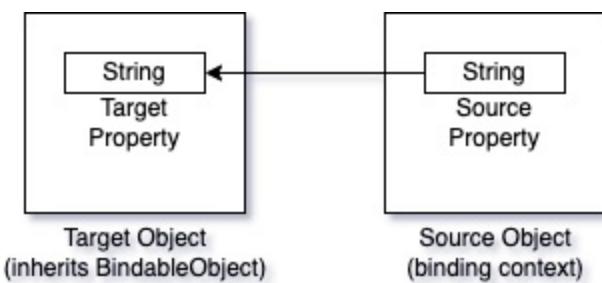
So far, we've been using our code behind to get values from and assign values to properties of UI element. By including the <http://schemas.microsoft.com/winfx/2009/xaml> schema in our XAML file,

we're able to use `x:... to assign names to our UI elements (such as x:Name="TodoTitleEntry"), and reference those UI elements by their assigned name in code. This lets us use TodoTitleEntry.Text to read the value of the Text property of the UI element called TodoTitleEntry.`

XAML is a markup language, but the UI elements we use in XAML are still classes. When we add an `<Entry... />` to our view, our view (which is also a class) at runtime instantiates an object of type `Entry`. If you examine the properties of the `Entry` class, you will see that it has a property called `Text` of type `string`, which is where we get the value the user has entered.

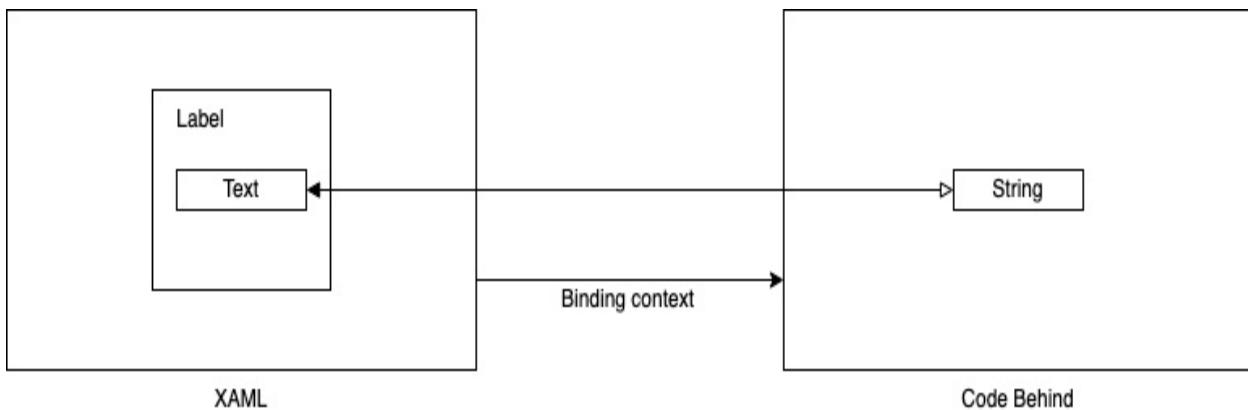
This on its own is powerful and gives us a neat way of getting and setting values from the UI. But a better way is to use **data binding**. Data binding lets us bind a property to another property, meaning that when one changes, so does the other.

Figure 3.14 A property on a class that inherits from `BindableObject` can be bound to a property of the same type



Data binding connects a source property and a target property, and the target property must belong to an object that inherits the **BindableObject** base class (all the built-in UI controls in .NET MAUI inherit `BindableObject`). The target property is bound to the source, so the target is where the binding is set. The source property can belong to an object of any type but must match the type of the target property. If a target and source property are not of the same type, a value converter can be used – we'll learn more about value converters later in chapter 8.

Figure 3.15 Each view (target object) can have a single binding context (source object). Properties of the target view are bound to properties of the same type on the source object.



Each view can have a single **binding context**, which is the source object. Once the binding is set, properties on that view (target) can then be bound to properties on the source by referencing the property name. Individual properties of the control can still be bound arbitrarily to any property of any other object, but this requires explicit referencing, and setting the binding context for a control means any of the control properties can bind to any of the source properties by just using the property name.

Controls vs Views

Sometimes it might seem like the terms ‘view’ and ‘control’ are used interchangeably. In some circumstances they can be, but for the sake of disambiguation, remember that views come in three categories: pages, layouts, and controls.

Page: A page is special kind of view that (usually) fills the whole screen and can be navigated to. A Page contains one child layout, which can in turn contain multiple other layouts.

Layout: A layout is a view that is used to arrange items on a screen. Examples of layouts that we’ve seen are `ScrollView` (which is technically a control, but behaves like a layout) and `Grid`.

Control: A control is a view that explicitly either displays something on screen or receives user input. A `Label` is a control, as is an `Entry`.

Binding context is inherited, and cascades from parent to child components in a view. Binding contexts can be explicitly defined for any view, but if you

don't explicitly define a binding context for a control in a `ContentPage`, the binding context will be the same as the `ContentPage`'s binding context. For example, if you add a layout or collection to a `ContentPage`, the layout will inherit the `ContentPage`'s binding context, as will any of its child controls. If you explicitly set the binding context of the layout or collection, then child controls will inherit this instead.

Figure 3.16 Binding context can be set at any level and is inherited by child components from that point down.

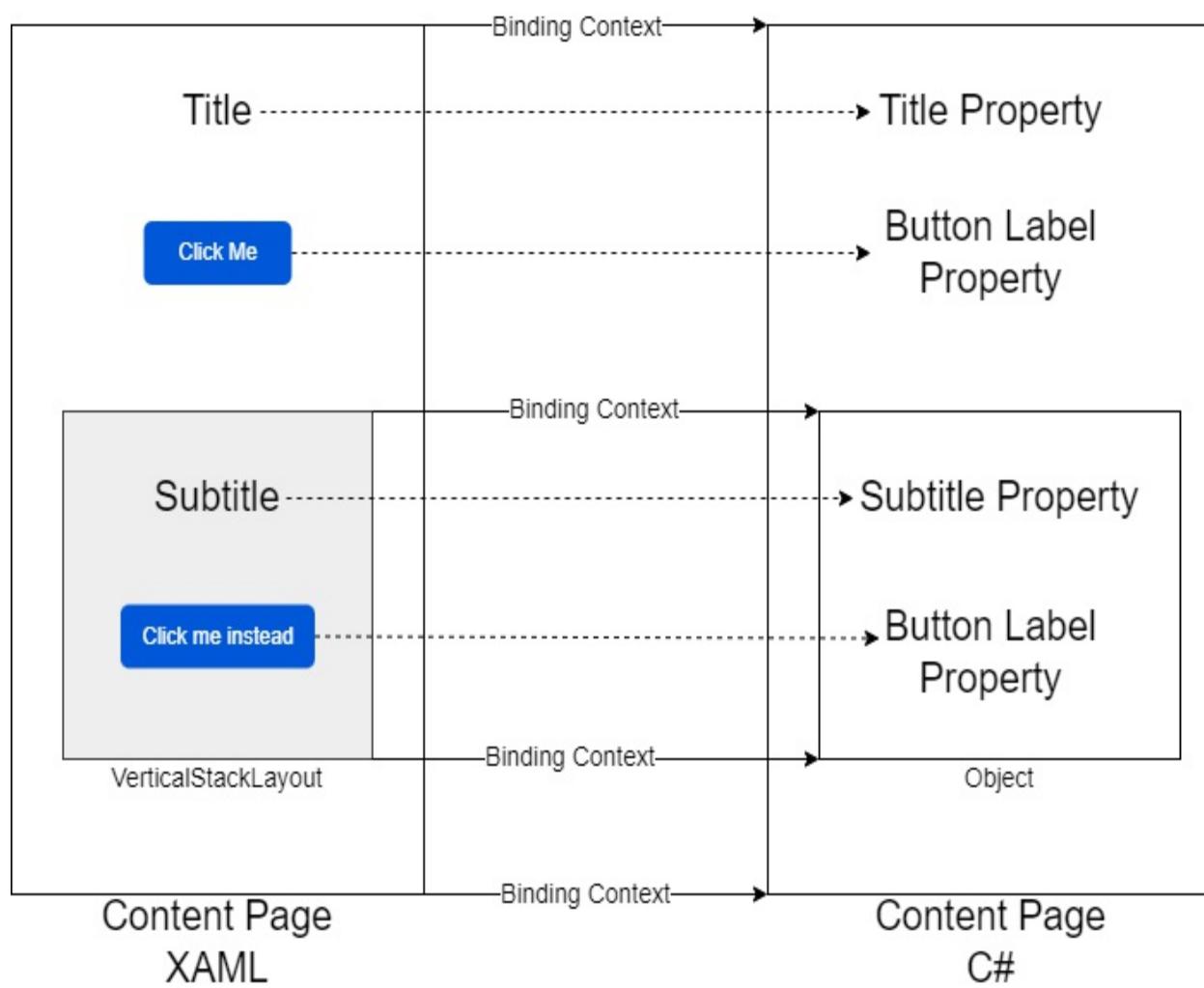
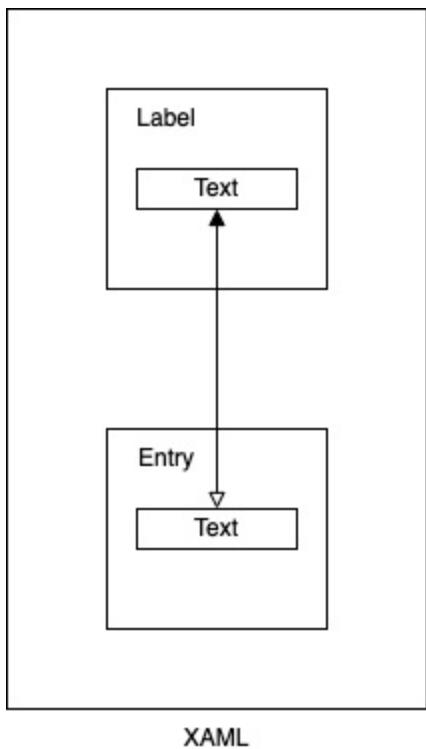


Figure 3.16 The binding context for this page has been set to the page's code behind. The `Title` and `Button` inherit this binding context and bind to properties on the `ContentPage` code behind. The binding context for the `VerticalStackLayout` has been set to an object property in the code behind.

The Subtitle and Button inside this layout inherit this binding context, and bind to properties on the Object. Understanding binding contexts is important, so we'll look at this in more detail later in this section.

Figure 3.17 Properties of views can be bound to properties of other views



Views can also be bound to other views. For example, `Label` and `Entry` both have a property called `Text` which is of type `string`, so these properties can be bound. By binding these two properties, we can have one automatically update the other.

3.4.1 View to View Bindings

View to view bindings can be set in XAML and enable us to bind properties in a view without any intervention in the code behind. Create a new .NET MAUI project called `Bindings`, and in `MainPage.xaml` delete the `ScrollView` and its contents.

Add the code from listing 3.12 to replace the `ScrollView` (so place within the `ContentPage` containing tags).

Listing 3.12 markup for view to view bindings

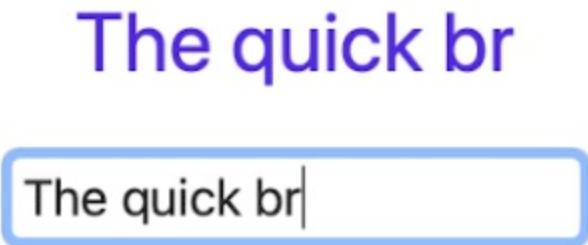
```
<VerticalStackLayout VerticalOptions="Center">#A
    HorizontalOptions="Center"
    Spacing="20"
    WidthRequest="200">

    <Label FontSize="Title"
        BindingContext="{x:Reference TextEntry}" #B
        Text="{Binding Text}"#C
        HorizontalTextAlignment="Center"/>

    <Entry x:Name="TextEntry"#D
        Placeholder="Enter some text..." />
</VerticalStackLayout>
```

Before running this project, delete the count field and the OnCounterClicked method from the code behind. Then run your project and start typing some text in the Entry. You should see something like figure 3.18 (depending on which platform you are running on).

Figure 3.18 the Text property of a Label bound to the Text property of an Entry. When the value of Text on the Entry changes, the label changes automatically.



Note how as you type, the text of the label changes. In previous examples, we've used the code behind to get properties from controls in the UI, and then set properties on other controls. With data binding, we just point one property at the other and the rest is taken care of for us.

View to view binding is useful when you have properties of the same type on two different views that are interrelated. In this example, we update a `Label` with the text entered in an `Entry`. Similar behaviour you may have seen in a real-world app would be updating numbers on a mock-up of a credit card as a user enters them. Another useful real-world example you may be familiar with is changing the zoom level or an image or map in response to a slider. Let's go ahead and add this functionality to our bindings app.

In the `MainPage.xaml` file of the Bindings app, add the two controls from listing 3.13 underneath the `Entry` in the `VerticalStackLayout`.

Listing 3.13 controls to add to the Bindings app

```
<Slider x:Name="ZoomSlider" />#A  
<Image Source="dotnet_bot.jpg"#B  
      WidthRequest="300"  
      HorizontalOptions="Center"  
      BindingContext="{x:Reference ZoomSlider}"#C  
      Scale="{Binding Value}" /> #D
```

Run the Bindings app, you should see something similar to figure 3.19.

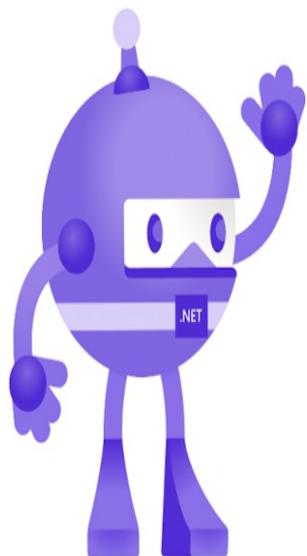
Figure 3.19 The Bindings app with the Scale property of an Image bound to the Value property of a Slider



Bindings

Binding is cool!

Binding is cool!



Try moving the `Slider` back and forth – you should see the scale of the image respond in real time. We've used data binding to bind the `Scale` property of the `Image` to the `Value` property of the `Slider`. When `Value` changes on the `Slider`, `Scale` changes on the `Image`.

We could have done this without data binding, but that would have been a much more convoluted process. We would have to create a method to be called by the `ValueChanged` event of the `Slider`, and with each change in value, set the `Scale` property of the `Image`.

3.4.2 Collections and Bindings in code

Data binding is an incredibly powerful feature of .NET MAUI, and it does much more than just remove boilerplate code. It opens a range of possibilities that we wouldn't have without it.

One of the most powerful uses of data binding is setting a binding source for a `CollectionView`, which is a property called **ItemsSource**. A `CollectionView` displays a collection of items, such as a `List<T>`, on screen. Collections or lists are the core of many apps, and with .NET MAUI, we can use data binding to define how we want each individual item in the collection to be displayed.

The most powerful feature of `CollectionView` is the use of a **DataTemplate**. A `DataTemplate` is the definition of how to display things on screen. In our current example we are using a `Label` which can only be used to display text. With a `DataTemplate`, we can define any layout we can imagine, and can bind aspects of that layout to properties on our binding context.

While a `DataTemplate` is a definition of how to display an item on-screen, an `ItemTemplate` is the specific `DataTemplate` in use for the `CollectionView`. `ItemTemplate` is a property of a `CollectionView`, and a `DataTemplate` is assigned to it. `DataTemplates` can be defined inline (as we will do in the following example), but can also be independent, reusable views. We'll see how to do this later in the book.

Rather than inherit the entire collection that the `CollectionView` is bound to,

the binding context for an `ItemTemplate` is the item itself. This means that for a collection of objects with a string property called `Name`, you can set the binding for the `Text` property of a `Label` in the `ItemTemplate` to `Name`, and it will display the desired value.

Looking back at our MauiTodo app, we can make a significant improvement to the way we display our to-do items. At the moment we have a `Label` which displays the title of every to-do item in our database, separated with a newline character. This is primitive and fairly limited; for example, we have no way to display the due date, or format different items based on their properties (like overdue, completed, etc.).

We'll update our app to use a `CollectionView`. This will let us define a specific layout for presenting each item in the to-do list, and we'll bind the `CollectionView` in code to an **ObservableCollection** of `TodoItems`. An `ObservableCollection` is a generic collection like a `List`, except it is automatically wired up through the XAML engine to provide a notification when items are added, removed, or updated, without us having to do any extra steps in the code. This will automatically update the `CollectionView` in the UI for us if we add or remove to-do items.

Let's start with the UI. In the `MainPage.xaml` file, get rid of the `ScrollView` on row 4 of the `Grid`, and its contents, and replace it with the `CollectionView` in listing 3.14.

Listing 3.14 The Todos CollectionView

```
<CollectionView Grid.Row="4"
    x:Name="TodosCollection">#A
<CollectionView.ItemTemplate>#B
<DataTemplate>#C
    <Grid WidthRequest="350">#D
        Padding="10"
        Margin="0, 20"
        ColumnDefinitions="2*, 5*"#E
        RowDefinitions="Auto, 50"#F
        x:Name="TodoItem">

        <CheckBox VerticalOptions="Center">#G
            HorizontalOptions="Center"
            Grid.Column="0"
```

```

        Grid.Row="0" />

<Label Text="{Binding Title}"#H
    FontAttributes="Bold"
    LineBreakMode="WordWrap"
    HorizontalOptions="StartAndExpand"
    FontSize="Large"
    Grid.Row="0"
    Grid.Column="1"/>

<Label Text="{Binding Due, StringFormat='{0:dd MM
#I
Grid.Column="1"
Grid.Row="1"/>
</Grid>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
```

As you can see, we've added a `CollectionView`, and provided a definition of how each item in the collection should be displayed, but we haven't bound it to a collection. We'll do this now in the code. Open the `MainPage.xaml.cs` file, and at the very top of the class (before the private member definitions), add an `ObservableCollection` of `TodoItems`.

```
public ObservableCollection<TodoItem> Todos { get; set; } = new()
```

`ObservableCollection` is in the `System.Collections.ObjectModel` namespace, so add a `using` statement to this at the top of the file. Then, in the constructor, set the `ItemsSource` property of the `CollectionView` to the `ObservableCollection`:

```
TodosCollection.ItemsSource = Todos;
```

Now we need to get rid of the code we're not using anymore. Delete the `private _todoListData` string, and every line that references it. This will be the assignment to it in the `foreach` loop in the `Initialise` method, assigning the value to the `Label` in the `Initialise` method, adding to it in the `Button_Clicked` method, and assigning it to the `Label` in the `Button_Clicked` method.

Now we need to add some logic back in to handle initializing the collection

and updating it when a user adds a new to-do item. Let's start with initializing – in the foreach loop in the Initialise method, add the to-do item in the loop to the ObservableCollection:

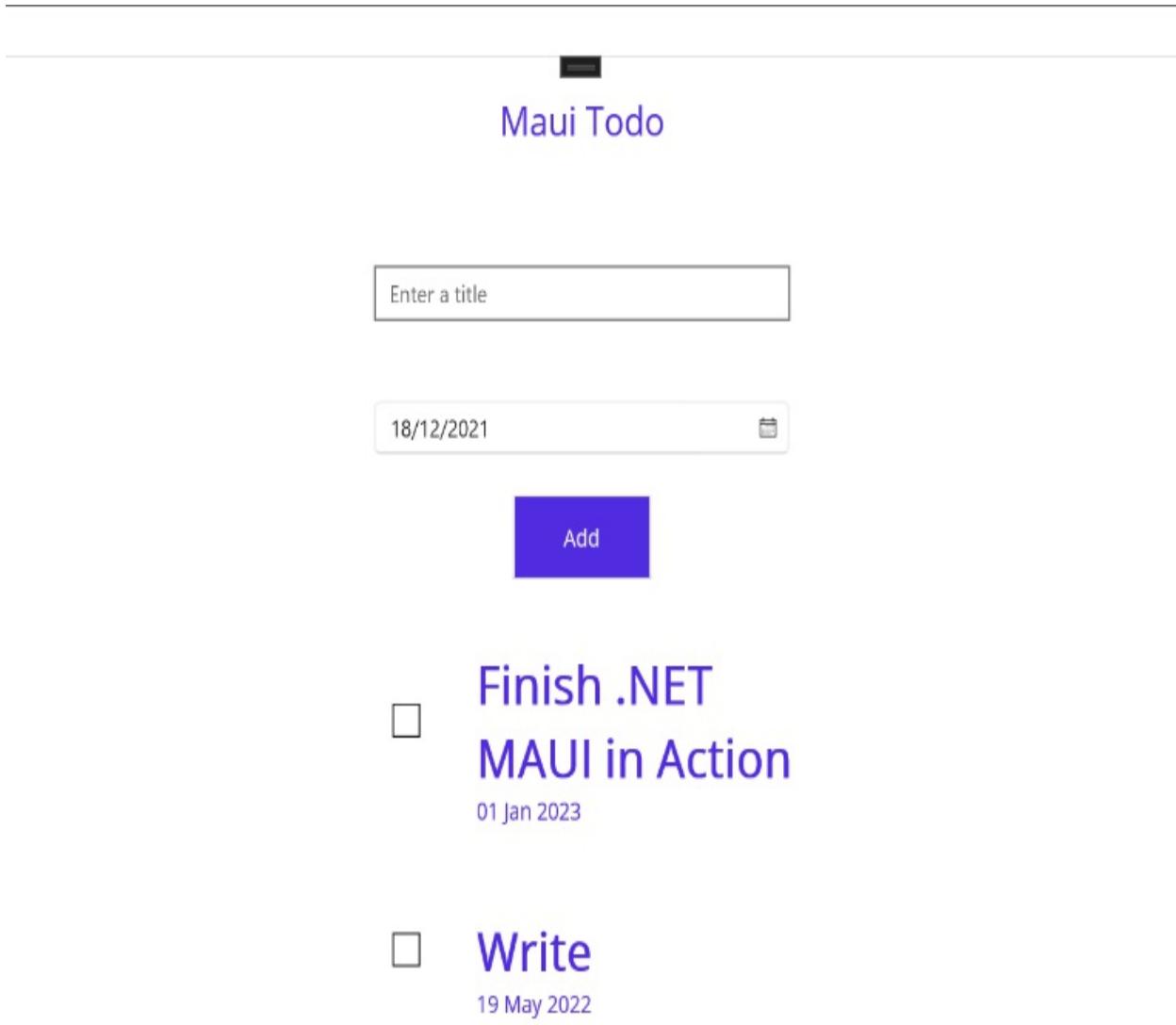
```
Todos.Add(todo);
```

And finally, add the same logic in the Button_Clicked method, in the if statement where previously we'd added the item to the string:

```
Todos.Add(todo);
```

That's all the changes we need now. To recap, we have added a CollectionView to the UI, and defined a data template, which specifies how items in the CollectionView should be displayed. We've added an ObservableCollection – a special type of collection that sends notifications when its content changes – to the code-behind, and bound it as the source of the items the CollectionView will display. And we've added some code to add to-do items to the CollectionView when the page initializes and when a user adds a new to-do item. Run the app and add some to-do items. You should see something like figure 3.20.

Figure 3.20 MauiTodo with a CollectionView running on Windows



3.4.3 ItemsSource bindings in XAML

Using code to set binding contexts is powerful, but given that databinding is what gives XAML its superpowers, we should have a look at how to set the items source for our CollectionView in XAML too.

We can set the binding context for any view directly in XAML, as we saw in section 3.4.2 when we set the binding context of one control to properties of another. Let's refactor the MauiTodo app to do all the binding in the XAML.

First, open the MainPage.xaml.cs code behind file, and in the constructor

remove the line we added in the previous section to set the items source:

```
TodosCollection.ItemsSource = Todos;
```

Instead, we'll set up this binding in the XAML. Open the `MainPage.xaml` file, and in the `<ContentPage...>` opening tag, add a name (so that it can be referenced) and a binding context. Then, in the `CollectionView`, bind the `ItemsSource` property to the `Todos ObservableCollection` in the binding context. Listing 3.15 shows the new opening `ContentPage` tag and the new opening `CollectionView` tag, with the added lines in bold.

Listing 3.15 The MainPage.xaml ContentPage tag changes

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiTodo.MainPage"
    BackgroundColor="{DynamicResource SecondaryColor}"
    x:Name="PageTodo"#A
    BindingContext="{x:Reference PageTodo}"#B

...
<CollectionView Grid.Row="4"
    ItemsSource="{Binding Todos}"#C
    x:Name="TodosCollection">
```

Run the app again. It should run without any problems, and you should see the same result (similar to figure 3.19). While there is no visible change to the UI, the improvement to the code is significant. Previously, we had the binding for the `ItemsSource` property of the `CollectionView` set in code, and all other bindings set in XAML. Now, the XAML is responsible for setting all the bindings.

This will become even more important when we learn about the MVVM pattern in chapter 7, and see how this approach can help us to maintain clean separation of concerns in our code.

3.5 Summary

- You can access common cross-platform device and OS features using

the .NET MAUI APIs. This gives you access to things like location, sharing, storage and many others.

- You must request permission from the user before accessing any privacy-sensitive features, like location for example, and you can request these permissions at runtime. In the app metadata, which is defined in a file specific for each platform, you declare which permissions you will request and for Android what features you will use.
- As part of .NET, MAUI gives you access to the entire NuGet ecosystem. This means you can leverage your existing .NET skills and use the packages you are already familiar with to build .NET MAUI apps.
- You can store data locally for use offline, which elevates your .NET MAUI app beyond a web page. You can persist key/value pairs with Preferences API, and can also save encrypted key/value pairs with the SecureStorage API.
- You can save more complex data than key/value pairs, including images, videos, or sound recordings, using the file system. When you pair this with NuGet, you can use a database like SQLite to store and retrieve complex data structures and can work completely offline.
- You can use code to retrieve values from, and set values to, elements in the UI. But with data binding, you simply connect properties, either between UI elements and other UI elements, or between UI elements and code. Once bound, changes in the source property will be automatically reflected in the target.
- You represent lists of complex models on-screen using a `CollectionView`. A `CollectionView` has a property called `ItemsSource` which, using data binding, can be bound to a `List` of items. `CollectionView` also has a property called `ItemTemplate`, and you assign it a `DataTemplate` which defines how each item in the collection should be displayed.

4 Controls

This chapter covers:

- What is meant by the broad term ‘View’
- The built in cross-platform controls that are abstracted for use in .NET MAUI apps
- How to display collections or lists of data using templated views
- How to use common modifiers to change the visual appearance of views
- How to use common modifiers to add or update functionality of views

Controls are views that either directly render something on screen (like an `Image` or a `Label`) or take input from a user (like a `CheckBox` or `DatePicker`). There are roughly 30 controls that ship with .NET MAUI out of the box, covering all the common use cases for UI applications, and we’ve already used several of these in our apps and seen how they work, such as `Image`, `Label`, and `Entry`.

In addition to the built-in controls, there are more available in the .NET MAUI Community Toolkit, as well as in free and commercial UI kits available that provide a range of additional controls, and stylised or customisable versions of the built-in controls too. And, of course, you can build your own controls too, which we’ll look at in chapter 8.

You can find a link to the .NET MAUI documentation in appendix A, which provides comprehensive coverage for each of these controls, including all their properties, events, and commands. I’m not going to repeat the documentation here; instead, I’ve broken the controls down into categories (grouped roughly the same ways as in the docs), and will provide a summary of each control, explaining what their use case is, what their primary data type is, and what the corresponding property name is. Throughout the book, and as we’ve already seen, you’ll learn how to use these controls *in situ*, rather than learning them by rote.

The documentation is an excellent source for the minutiae of each control’s

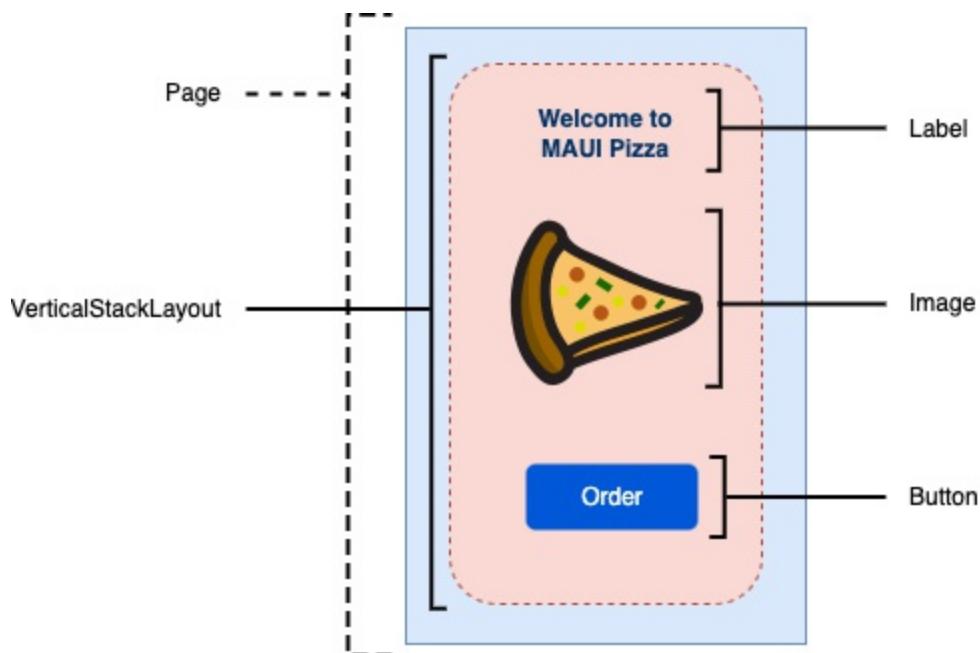
individual properties, and, if you’re using Visual Studio, IntelliSense is also an invaluable tool. Simply dropping a control onto your page or layout, using either XAML or C#, will immediately open up all the control’s properties for you to explore.

While I won’t cover all the details of every control here, there are some properties that are common across all controls. Additionally, there are some supplementary controls that I call control modifiers which can be applied to any control. I’ll go through the shared properties and control modifiers that you’re likely to find most useful in your .NET MAUI apps.

4.1 What do we mean by ‘Views’?

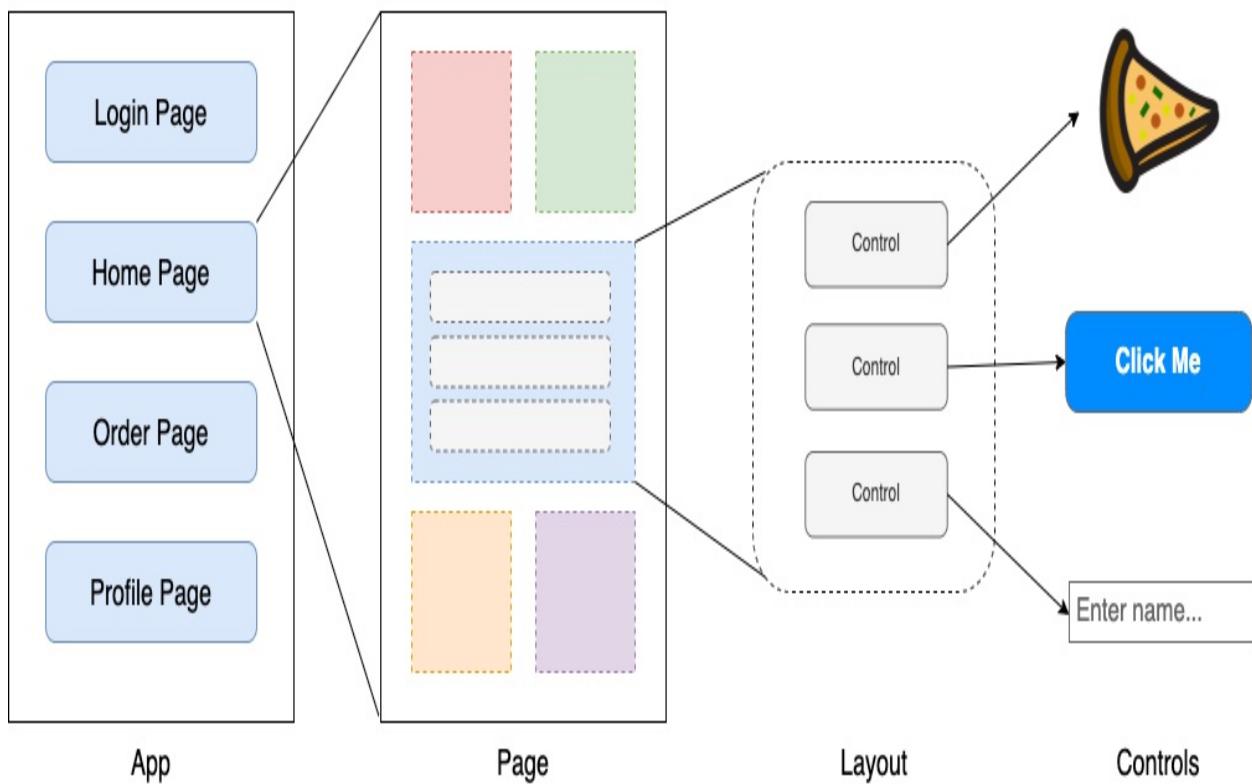
Views in a UI are the things that are displayed on screen. Unlike other parts of an app (say a service for example), these are the things that your users see and interact with.

Figure 4.1 Everything on screen in a UI app is a view, and must be contained within a page (which, being on screen, is also a view). A page takes up the full screen or window and can be navigated to or instantiated. In .NET MAUI, a page can only contain one child, in this case a VerticalStackLayout, which displays its child items vertically. Layouts can contain an arbitrary number of child items, which can either be controls, like Buttons or Labels, or other layouts like Grids or HorizontalStackLayouts.



In a .NET MAUI app, there are three types of view: *controls*, *layouts*, and *pages*. In the hierarchy of a .NET MAUI app, a Page is something that the app itself can display. A .NET MAUI app doesn't know how to directly render a layout or a control, but it does know how to display a Page on the screen.

Figure 4.2 An App knows how to render Pages, and Pages know how to display a single layout. Layouts define how to arrange controls or other layouts on the screen. In this example, an App renders a Home Page which has a Grid layout as a child. The Grid layout here has five child items, including a VerticalStackLayout in the middle. The VerticalStackLayout has three controls as child items: an Image, a Button and an Entry.



Important note on terms

In the official .NET MAUI documentation, ‘control’ is used as an umbrella term for everything you see on screen, including pages, layouts and interactive and display elements like buttons and images, while ‘view’ is used to describe those elements.

To make the concept easier to understand, I have switched these around here

and used the industry's commonly used definitions instead: 'view' for the umbrella term, and 'controls' for the interactive and display elements (the definition of pages remains the same).

It's important to know what the official terms are, especially when referring to the documentation, and I normally use these in this book. However, I made an exception here as I feel that the concepts are easier to understand using the standard terms, especially when we come to the discussion of the MVVM pattern later in the book.

In this chapter, we'll look specifically at controls. We'll look at layouts and pages in the coming chapters.

4.2 Cross-platform Controls

.NET MAUI provides a collection of controls out of the box that cover a broad spectrum of use cases for mobile and desktop apps. These controls are abstractions of their native platform implementations, as discussed in chapter 1, meaning that when you want to accept text input from a user, you use an `Entry`, that looks like an iOS control on iOS, a Windows control on Windows, etc., without having to implement the same control once for each platform.

This is the simple consistent but not identical approach we discussed in chapter 1. You can of course make these controls look identical on each platform which, we'll discuss in chapter 11. This section provides a brief overview of these controls and their use cases.

You will already have seen how to use some of these from the apps we've already built. Others we will use in future apps, but in general these are very simple to consume in your apps, and you can find full coverage for all of these in the .NET MAUI documentation.

4.2.1 Displaying Information

The controls listed in this section are used for presenting information to your user. That means that these controls do not accept user input. We've already

used the `Label` control; table 4.1 summarises its usage as well as the other controls in this category.

Table 4.1 A summary of the .NET MAUI controls for displaying information to the user

Control	Primary data type	Primary data type name	Use case
Label	String	Text	Used for displaying text on screen. We've used Labels in every app we've looked at so far; it's almost impossible to build an app without them!
ProgressBar	Double	Progress	Used to display a value expressed as a fraction. Usually used to indicate progress, e.g., how much of a file has downloaded.
ActivityIndicator	Bool	IsRunning	Used to show that something is happening. Different from a ProgressBar in that it just shows that an activity is happening, rather than giving an indication as to the completion status of that activity.

4.2.2 Accepting Input

.NET MAUI provides several controls that allow you to accept input from your user. Each of these have a property that represents the user's selected value. These properties are all bindable, so you can either use data binding, or

read the value of the property from the control directly in your code behind.

Table 4.2 lists each of these controls, along with their value property and use case.

Table 4.2 the controls provided by .NET MAUI for accepting input

Control	Value property	Primary data type name	Use case
Entry	Text	String	Allows a user to enter text
Editor	Text	String	Allows a user to enter text across multiple lines
CheckBox	IsChecked	Bool	Provides a yes/no, true/false, or on/off option.
DatePicker	Date	DateTime	Allows the user to select a date
Slider	Value	Double	Allows a user to select a value between a minimum and maximum (by default between 0 and 1)
Stepper	Value	Double	Increases or decreases a number by a specified amount (1 by default)
TimePicker	Time	TimeSpan	Allows the user to select a time

RadioButton	IsChecked	Bool	Allows the user to select one option from a group of displayed options by checking the box
Picker	SelectedItem	Object	Allows the user to select one option from a group of listed options by picking it from the list

Note that just like the other controls in this chapter, each of these has many more properties than are listed here, some of which we'll cover as we progress through the book. It's also easy to discover these properties through Intellisense as you're coding, which I encourage you to do with all controls, but especially these. Most of them are self-evident, but full coverage is available in the .NET MAUI documentation.

4.2.3 Accepting Commands

.NET MAUI offers a collection of controls that allow you to accept commands from your user. These controls differ from controls that accept input (see next section) in that these controls strictly tell your app that the user wants to do something, as opposed to your user providing a value.

Each of these controls has a bindable property of type `ICommand` that you can use to execute an action in response to the user interacting with the control. They also have event handlers that you can use in your code behind file without binding. Table 4.3 shows the various command inputs and their use cases.

Table 4.3 The controls in .NET MAUI that initiate an action or command. The `ICommand` property of each control is bindable, or you can use the event handler instead.

Control	<code>ICommand</code> property name	Event handler	Use case

Button	Command	Clicked	A simple control that user can tap or click on to initiate an action. Uses text to convey its intent.
ImageButton	Command	Clicked	Same as Button, but uses an image to convey its intent instead of text
SearchBar	SearchCommand	SearchButtonPressed, TextChanged (inherited from Input)	Presents a recognisable search box. It has text input and a magnifier icon. The SearchCommand ICommand and SearchButtonPressed event handler are both linked to the magnifier icon. Any of the commands or event handlers can be linked to code that searches or filters a list.

Note that in the .NET MAUI documentation, two additional controls are grouped into this category. These are RadioButton, which I cover under accepting input, and SwipeView, which I cover in control modifiers.

4.2.4 Displaying Graphics

.NET MAUI provides three ways for presenting graphics: `Image`, `Shapes` and `GraphicsView`. The `Image` control allows you to display a graphical image (like a Jpeg, PNG or SVG file) from a variety of sources, and we've used this control in our apps already. The other two, `Shapes` and `GraphicsView`, allow you to draw directly on the screen programmatically.

Table 4.4 A summary of the controls available in .NET MAUI for displaying graphics

Control	Use case
Image	Used to display images. The <code>Source</code> property can accept images from a file, a resource, a URL or a Stream, and is populated automatically using static constructors (meaning you only need to supply the <code>source</code> property, and the type is inferred). We've used <code>Image</code> s in a couple of our apps already, using the <code>Resource</code> image type, and we'll use URL later in this chapter.
Shapes	The <code>Shapes</code> API lets you either draw simple geometric shapes on screen, or lets you render complex images with an SVG compatible <code>Path</code> control. It's useful for quickly placing a simple shape onto your page.
GraphicsView	<code>GraphicsView</code> is new to .NET MAUI and part of the <code>MAUI.Graphics</code> library. <code>GraphicsView</code> exposes a canvas that lets you draw simple or complex images on screen using brushes and paths. We'll look at this cool feature in chapter 8.

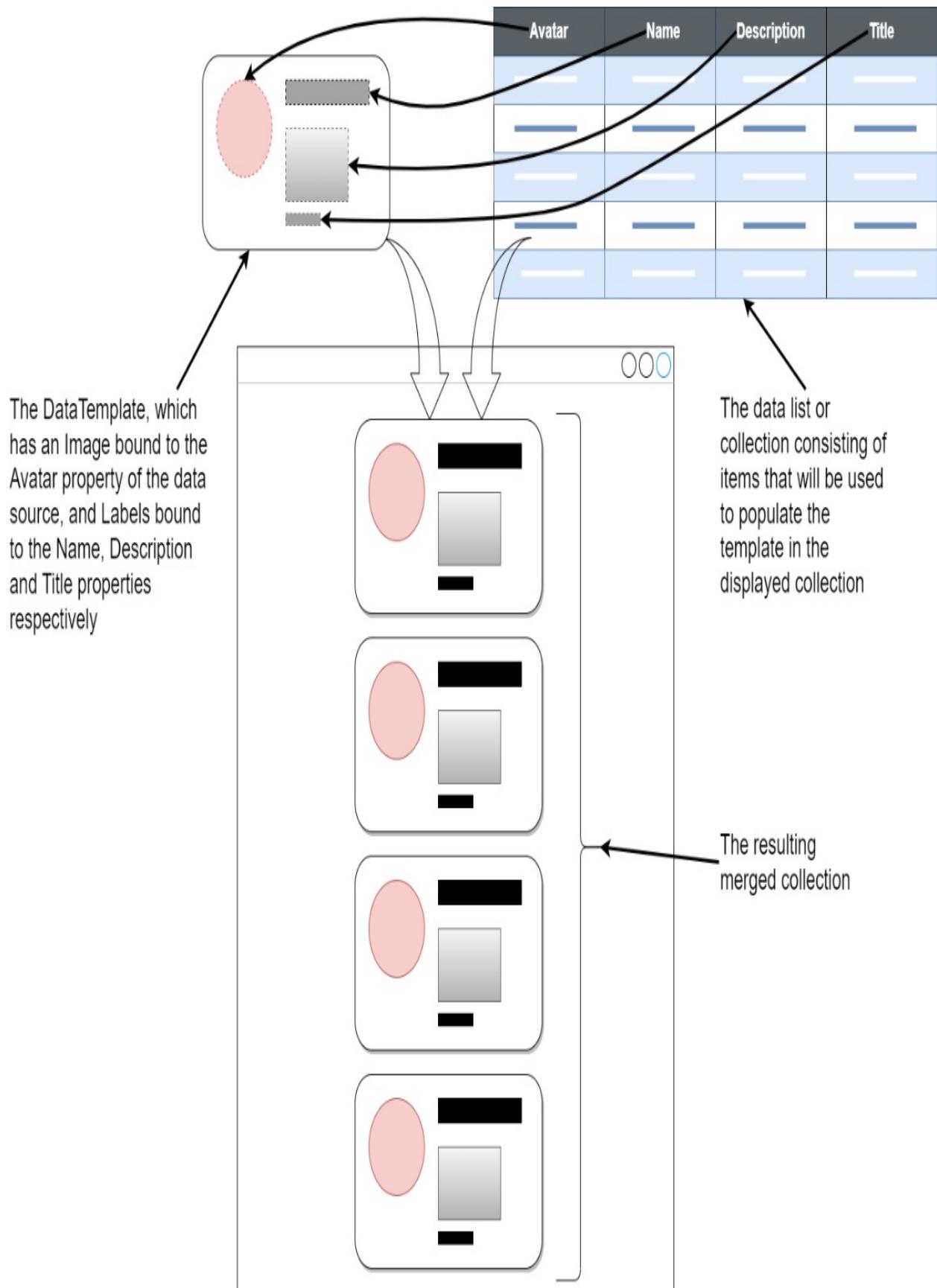
4.3 Displaying Lists and Collections

.NET MAUI provides a few controls that are useful for displaying collections of data. We have already used one of these controls, `CollectionView`, in the MauiTodo app, to display a collection of to-do items, using a `DataTemplate` to define how each to-do item should be presented.

The most efficient way to display a collection of data in your app is to define a template for each item in the collection, and then let the app render out the collection based on your template.

There are a number of controls in .NET MAUI that provide this functionality, and while they have different use cases and the way that they arrange items in the collection differs, they share some similarities, and three of them have common properties for declaring the collection of data and the template.

Figure 4.3 Templated views take a template and a collection of data, and render each item in the collection according to the template



The key properties of these controls are `ItemsSource` and `ItemTemplate`. `ItemsSource` is a bindable property that can be bound to a collection of type `IEnumerable<T>` (or any inherited collection type). This makes it easy to set these bindings in XAML, and while you can't *directly* assign a collection to this property in code, you can use the `SetBinding` method, inherited from `BindableObject`, if you want to set the binding in code. We'll use XAML bindings in this book, but you can refer to the documentation if this is an approach you want to follow.

`ItemTemplate`, as the name suggests, defines how each item in the collection will be presented. The collection-based view uses the template you provide and renders each item in the collection according to the template. As we saw in MauiTodo, we assign a `DataTemplate` to the `ItemTemplate` property of a collection control.

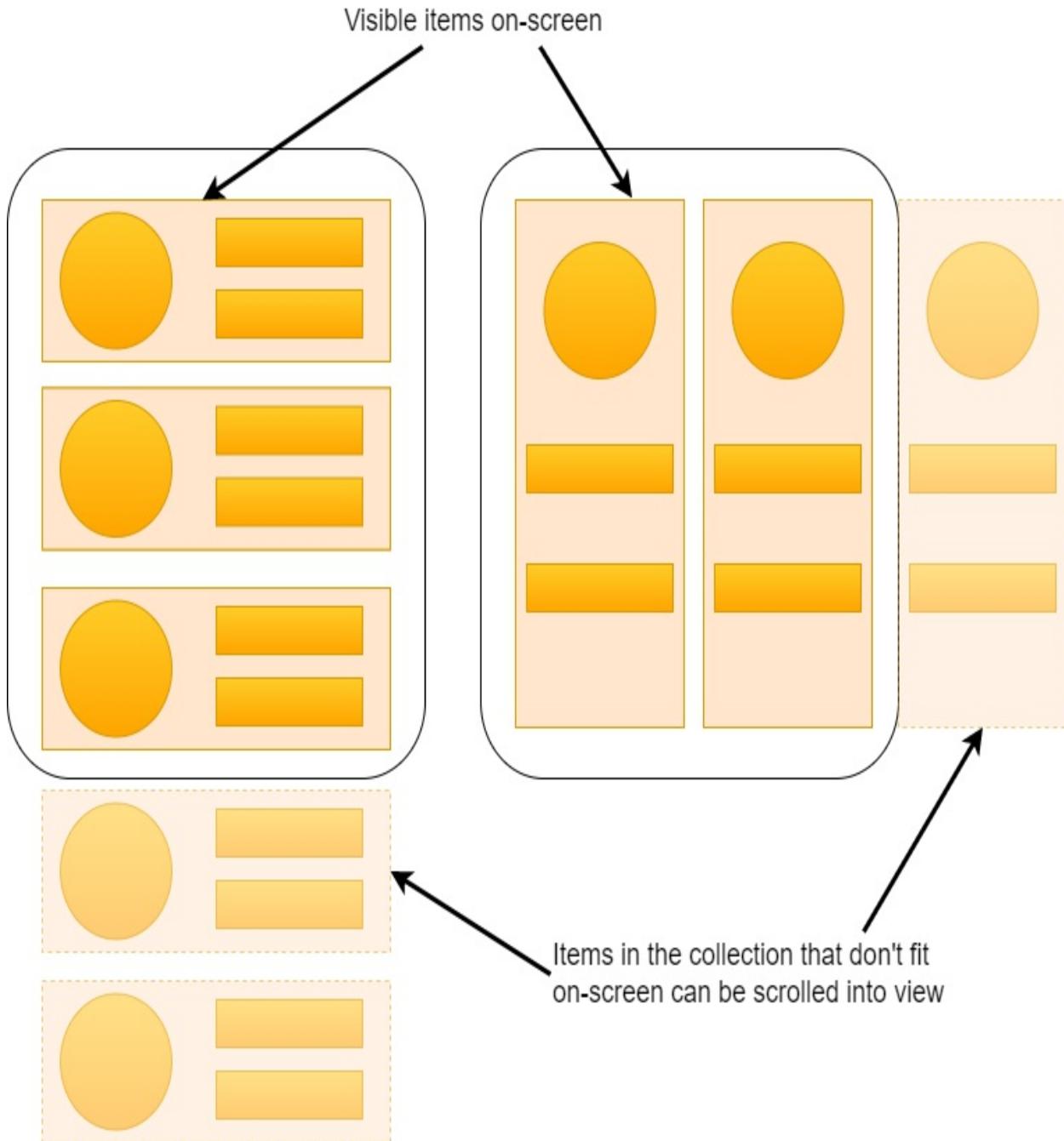
As well as statically defining a single item template, we can also use a data template selector to choose different templates at runtime based on properties of each item in the collection.

One of these controls, `TableView`, differs from the other three in this category in that it doesn't have an items source property. `TableView` doesn't use a template or a data source; instead, you add child views statically. The following sections summarise of the collection views available in .NET MAU and their potential use cases.

4.3.1 CollectionView

As mentioned, we have used `CollectionView` already in MauiTodo, and we will use it again in later examples. `CollectionView` is arguably the workhorse of .NET MAUI apps and likely something you will use a lot.

Figure 4.4 CollectionView shows a scrollable list of templated items, based on a collection provided by a data source. It can be oriented either vertically or horizontally and will scroll automatically. It provides a hard stop when you reach the end of the collection.



`CollectionView` supports multiple layout types. In figure 4.4, we can see two of these: vertical list and horizontal list. `CollectionView` also supports vertical grid and horizontal grid layouts. Vertical list is the default, so you can use this layout by instantiating a `CollectionView`, without specifying any layout parameters.

Like the other controls in this section (except for `TableView`),

`CollectionView` has a bindable property called `ItemsSource` that provides the list of items for the `CollectionView` to render. The `CollectionView` has a `DataTemplate` that defines how each item should be presented.

There's lots more to learn about `CollectionView`, particularly with regard to controlling `ItemsLayout`, which we've only slightly touched on. And, more importantly, around items selection. `CollectionView` allows you to specify whether users can select one or multiple items in the collection, and provides bindable properties called `SelectedItem` and `SelectedItems` to support each mode respectively. We'll revisit both of these in `MauiTodo` in the coming chapters.

We'll use `CollectionView` again throughout this book, and I encourage you to experiment as much as possible, consulting the documentation where necessary.

4.3.2 ListView

`ListView` is conceptually similar to `CollectionView`, although it is focused on presenting items using one of a set of predefined templates (`TextCell`, `ImageCell`, `ButtonCell` and `SwitchCell`). Whereas with `CollectionView`, your `DataTemplate` can be an arbitrary layout of your own design, in `ListView` your `DataTemplate` must reference one of these cell types. There is however an additional cell type called `ViewCell` which would let you define a custom layout for items.

Another difference between `ListView` and `CollectionView` is that `CollectionView` supports multiple selection (you can select more than one item in the collection), whereas `ListView` supports single selection only. Another difference is that `ListView` supports context actions, which `CollectionView` does not; although you can use the `SwipeView` modifier control (covered later in this chapter) to add this functionality to items in a `CollectionView`.

Anything you can achieve with `ListView`, you can also do with `CollectionView`, but `CollectionView` provides better flexibility. There may be cases where you want to display a collection exclusively of `SwitchCell`,

for example, as well as needing context actions. In this case it could be easier to use `ListView`, but I still recommend using `CollectionView` instead.

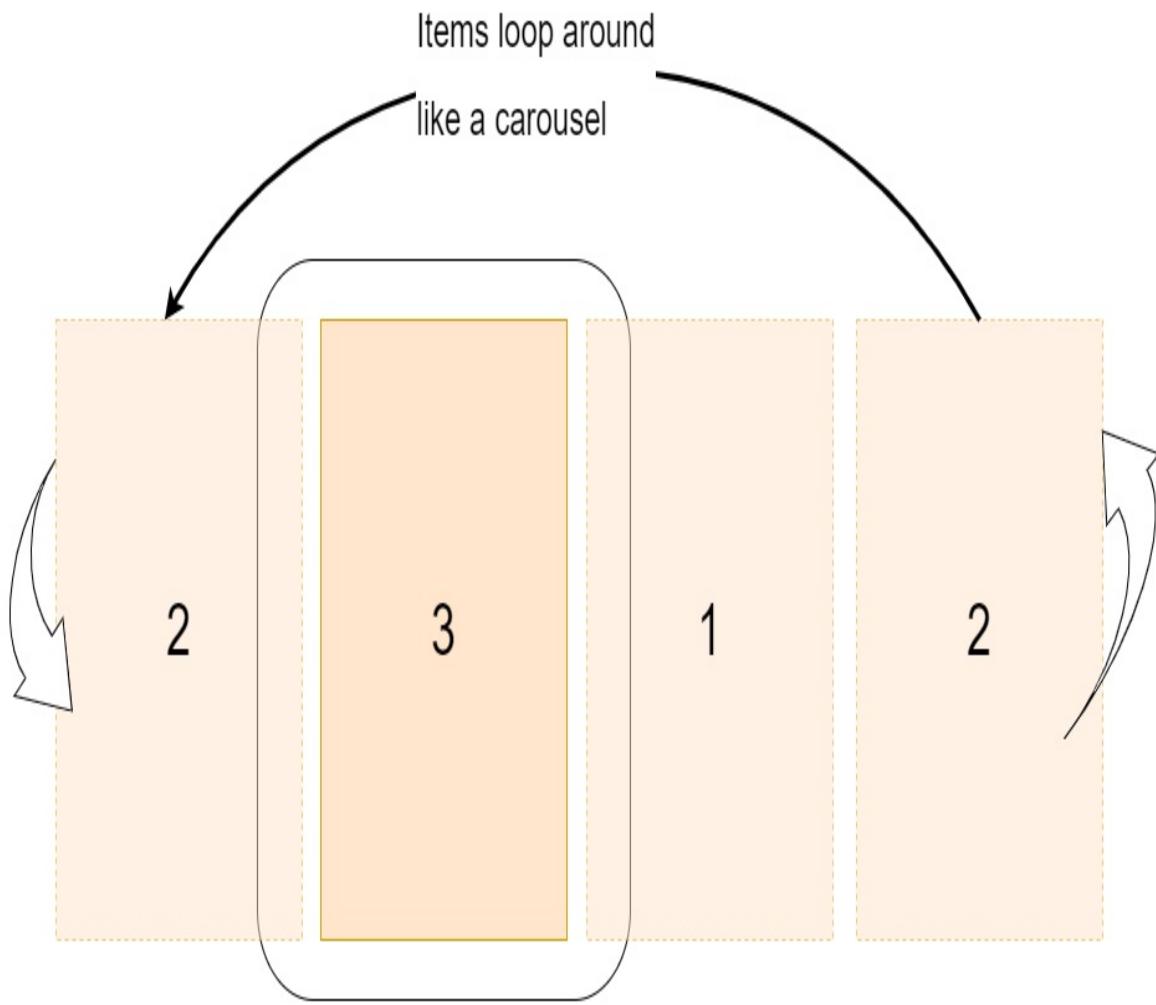
`CollectionView` was introduced as a more mature version of `ListView`, and while `ListView` is still available, `CollectionView` is what you should use in .NET MAUI.

4.3.3 CarouselView and IndicatorView

`CarouselView` is similar to `CollectionView`; in fact, it's an extension of `CollectionView` and is built on top of it. It binds to a collection of items and renders each of those items based on a template, in a horizontally or vertically scrolling sequence. However, there are some subtleties in their implementation that make them naturally lend themselves to different scenarios.

`CarouselView` defaults to a horizontal orientation, which can help to focus on a single item at a time, whereas the focus of a `CollectionView` (or `ListView`) is to display the collection itself.

Figure 4.5 `CarouselView` shows a scrollable list of templated items, based on a collection provided by a data source. Unlike a `CollectionView`, can be configured to return to the beginning when you reach the end (like a carousel). `CarouselView` can be oriented vertically or horizontally. And, unlike a carousel, you can scroll in both directions.



However, the main distinguishing feature of `CarouselView` is that it can return to the beginning when you reach the end of the collection displaying the items in a carousel, rather than a one-dimensional list (hence the name). You've probably seen carousels like this in websites; think of a scrolling section that shows the same three or four cards in a looping sequence.

Unlike a `CollectionView`, `CarouselView` does not allow selection of items per se; it does however offer a bindable property called `CurrentItem` which can be used to determine which is the currently focused item from the collection.

Let's see a `CarouselView` in action. **Download the CellBoutique app from the book's online resources.** Note in the `Models` folder there is a class called `Product`, which has four properties:

- **Title**: the name of the product
- **Description**: a brief description of the product
- **Price**: the product's price
- **Image**: a URL pointing to an image of the product

If you look at the code in `MainPage.xaml.cs`, you'll see that we've overridden the `OnAppearing` method, and in this method we are populating an `ObservableCollection` of type `Product` with a few items. Note also that in the constructor the page's binding context has been set to itself, so we can bind to properties of the page in the XAML.

In `MainPage.xaml`, we'll add a `CarouselView` and bind its `ItemsSource` property to this `ObservableCollection`, and we'll define a `DataTemplate` that will show all four of these properties. We'll also specify the `ItemsLayout` property so that we can specify some spacing between individual items.

`Listing 4.1` shows the code you need to add to `MainPage.xaml`, between the `<Content...>...</Content>` tags. The added code is shown in **bold**.

Listing 4.1 The updated `MainPage.xaml` from the `CellBoutique` app

```
<?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="CellBoutique.MainPage">

    <CarouselView ItemsSource="{Binding Products}"#A
        Margin="10">#B
        <CarouselView.ItemsLayout> #C
            <LinearItemsLayout Orientation="Horizontal"#D
                ItemSpacing="60"/#/E
        </CarouselView.ItemsLayout>
        <CarouselView.ItemTemplate>#F
            <DataTemplate>
                <VerticalStackLayout Spacing="20">#G
                    <Image Source="{Binding Image}"
                        WidthRequest="400"
                        HeightRequest="500"
                        Aspect="AspectFill"
                        HorizontalOptions="Center"/>
                    <HorizontalStackLayout Spacing="20">
```

```

<VerticalStackLayout WidthRequest="200"
    Spacing="10">
    <Label Text="{Binding Title}"
        FontAttributes="Bold" />
    <Label Text="{Binding Description}"
        LineBreakMode="WordWrap"/>
</VerticalStackLayout>
<Label FontAttributes="Bold"
    HorizontalTextAlignment="End"
    Text="{Binding Price, StringFormat='{}{0:C}'}"
/>
</HorizontalStackLayout>
</VerticalStackLayout>
</DataTemplate>
</CarouselView.ItemTemplate>
</CarouselView>
</ContentPage>

```

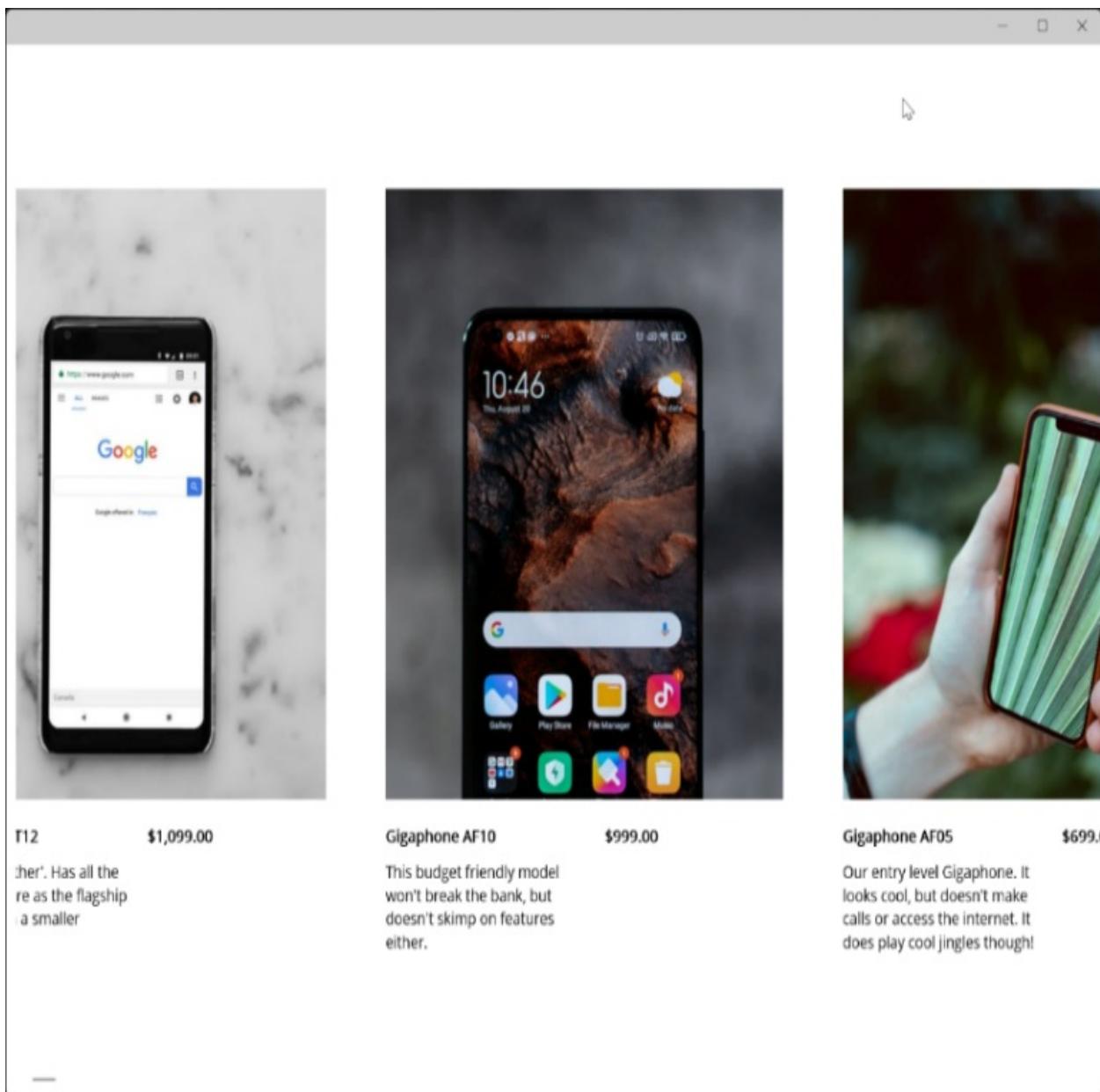
ImageSource in .NET MAUI

We've already seen how to display an image in a .NET MAUI app. There's one in the template, which we've replaced with our own .NET MAUI bot, and we've also imported an image to use in the FindMe! app. The `Image` control has a property called `Source` of type `ImageSource`. `ImageSource` has four methods that allow you to display an image from different sources, called `FromFile`, `FromUrl`, `FromResource` and `FromStream`.

`Source` is a bindable property, so you can either set the value directly, as we have done in previous examples, or use a binding, as we did in listing 4.1. When you use the `Image` control in XAML, .NET MAUI is smart enough to know what type of `ImageSource` you are using and will display the image accordingly. If you have your own product photos you want to use, feel free to import them into the CellBoutique app (if you recall from earlier examples, they go into the `Resources/Images` folder) and change the `Image` property on a couple of the `Product` entries. You'll see your local image displayed instead of one loaded from a URL without having to make any other code changes.

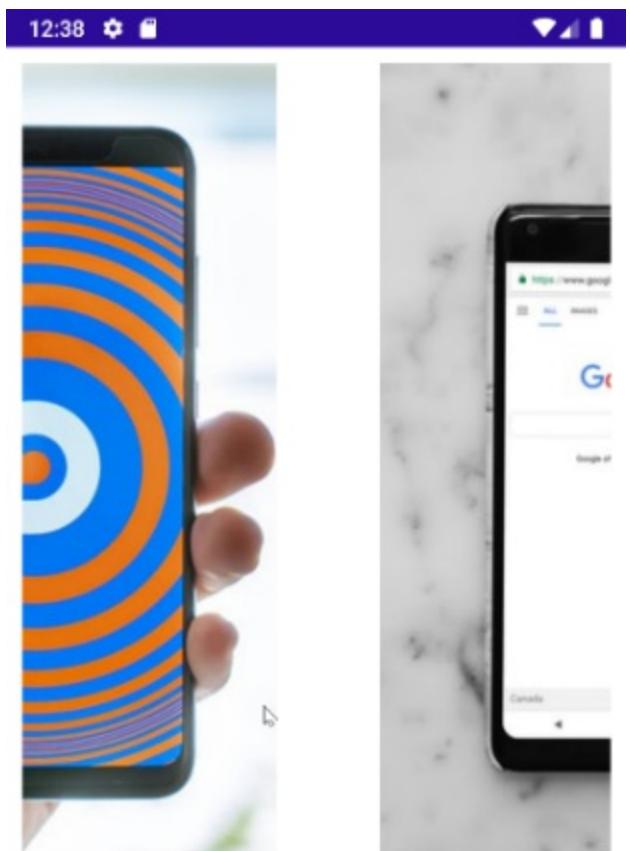
Go ahead and run the app, and you should see something like figure 4.6.

Figure 4.6 The CellBoutique app using CarouselView running on Windows. Items are displayed in a scrollable ‘carousel’, meaning the list repeats on a loop as you scroll through it. You use a scroll wheel or arrow keys to scroll through items on a desktop OS.



When using CarouselView on Windows (as in figure 4.6) or macOS, you can navigate through the carousel items either using the scroll wheel on your mouse or using the arrow keys on your keyboard. If you have a touchpad, some support swipe gestures which work here too.

Figure 4.7 The CellBoutique app running with CarouselView on Android. Items are displayed in a scrollable ‘carousel’, meaning the list repeats on a loop as you scroll through it. You use a swipe gesture to scroll through items on a desktop OS.



\$1,199.00

2
X50

Gigaphone XT12

The 'little brother'. Has same hardware as the XT16 but with a smaller screen.



When using `CarouselView` on Android (as in figure 4.7) or iOS, you can use a swipe gesture to navigate between items in the stack.

4.3.4 TableView

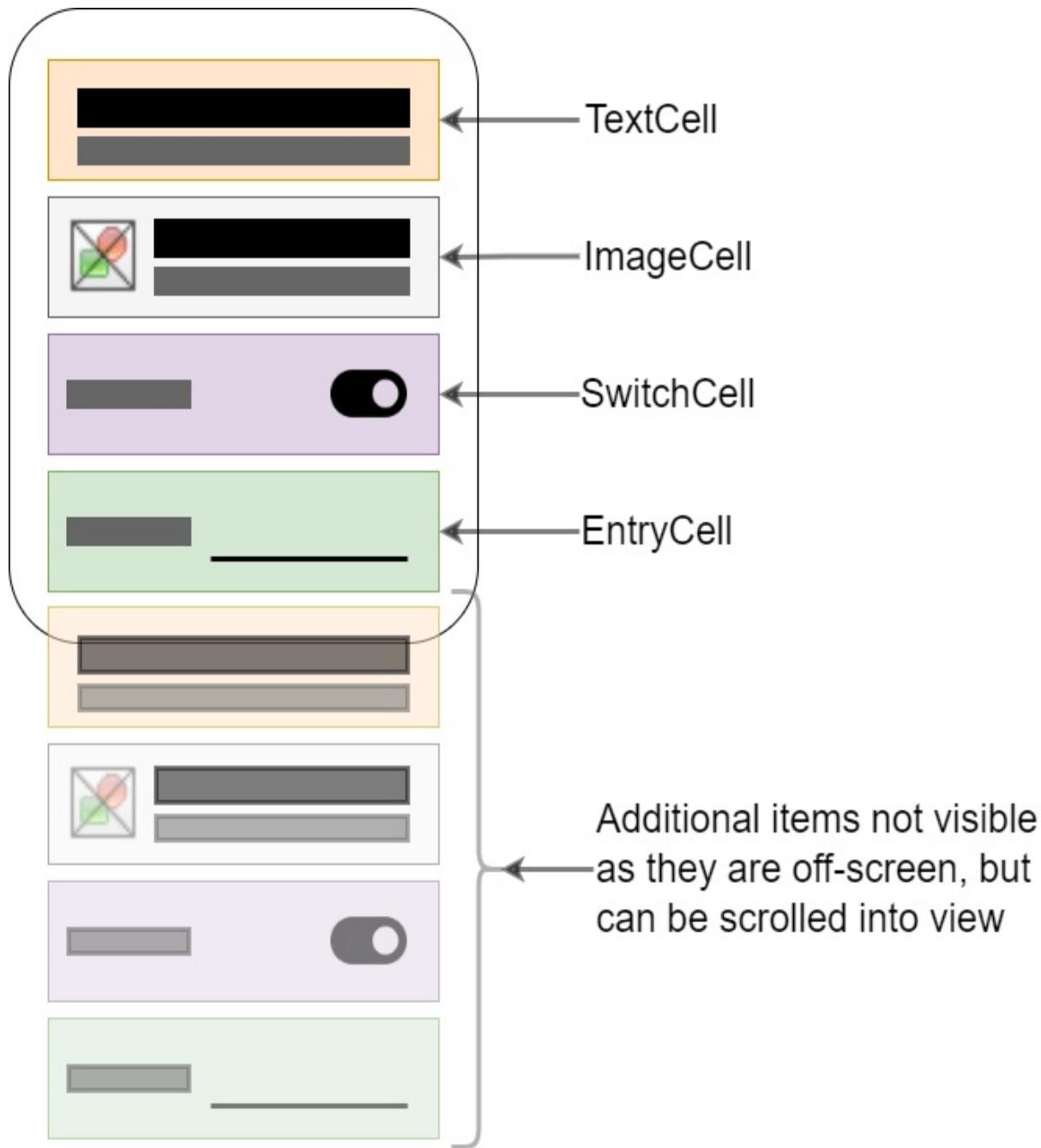
Unlike the other controls in this section, the child items displayed by `TableView` are added directly rather than being rendered automatically from a

source list or collection. `TableView` does not have an `ItemsSource` property.

Like `ListView`, with `TableView` you must use specific types of cells to display content. In `TableView` these cells are:

- `TextCell`: For displaying two `Labels`, a primary and secondary
- `ImageCell`: Like a `TextCell` but adds an image
- `SwitchCell`: For displaying text and a `Switch`
- `EntryCell`: For displaying a `Label` and editable text
- `ViewCell`: For displaying a custom layout

Figure 4.8 `TableView` shows a scrollable list of static, non-templated items. Each item in a `TableView` can be different and unrelated and is entered manually rather than displaying items in a collection from a data source.



The documentation suggests four use cases for `TableView`: a menu, a form, presenting data and settings. You will likely find that forms and settings are what you will use `TableView` for.

The best use case for `TableView` is for providing a settings page or screen that is consistent with the native OS settings app. While it's becoming more popular to create a unique, app themed or branded settings page (you can

read about this approach in a Microsoft blog post by James Montemagno, here: <https://devblogs.microsoft.com/xamarin/great-looking-settings-screens-for-xamarin-forms/>), an OS consistent settings page is still the dominant choice (see, for example, the Messenger app by Meta or Outlook by Microsoft). `TableView` is best suited for these kinds of settings pages.

4.4 Common Properties and Control Modifiers

There are few (if any) limits in terms of what modifications you can make to controls in .NET MAUI, but there are some common modifications you can make to any control (or layout, if it makes sense) using simple built-in tools.

The common modifiers we will look at here are:

- Clipping
- Borders
- Shadows
- Gesture recognisers
- SwipeView
- RefreshView

In this section, we'll look at these and see how some of them can be used to enhance MauiTodo. We'll also look at the two most used properties which are common to all views: height and width.

4.4.1 Height and Width

In a .NET MAUI app, all views have a `Height` and `width` property. These properties are read-only and can be used to get the height or width of a view. This is useful if you are building a responsive layout or need to adjust the size or position of a visual element in response to another.

As they are read-only, you can't use `Height` and `width` to set the height or width of a view, only to get it. Instead, we specify `HeightRequest` and `WidthRequest`, which are specified on the `VisualElement` base class that all controls and layouts inherit. These are values specified in device-independent units (DIUs).

Device-independent units

Working directly with pixels is impractical for modern UI development because every screen has a different resolution. And worse, even screens with the same resolution can be vastly different sizes. For example, it's not uncommon to find a mobile phone with a 1080p screen (or even 4k), with a diagonal screen size of 6 inches or less. This could be the same resolution as a 55 inch, or even 85-inch, TV screen.

Back when I used to write games on my Amiga 1200, I didn't have this problem. Every screen that my games would run on would reliably have a resolution of 320 x 240, so I could comfortably use precise pixel coordinates to place a sprite on screen and be confident that it would appear in the same position for everyone who ran it.

We don't have that luxury anymore, but fortunately the problem has been solved by the introduction of device-independent units (DIUs). They have slightly different names on different platforms (e.g., device-independent pixels, density-independent pixels, or points), but most platforms conform to the same sizing convention, which is roughly equivalent to 160 units per inch, or 64 units per centimetre.

This means if you give a size value of 2 for spacing in a grid, that means the space will be approximately 1/32 of a centimetre or 1/80 of an inch, irrespective of the size or resolution of the display.

Note my use of the words 'roughly' and 'approximately'; the system is not perfect (see note on the pixel perfect myth), but close enough in most cases to give you the desired UI effect.

Whenever you see a value for height, width, or thickness in a .NET MAUI app, whether in this book or any code you look at, if the size is not proportional (specified with an asterisk (*)), then the size is in DIUs.

When a layout calculation is triggered (e.g., when a page loads), a method called `GetSizeRequest` will calculate the layout bounds based on a combination of factors, including the requested height and width, and any other constraints such as the height and width of the parent element.

`HeightRequest` and `WidthRequest` are good enough to give us the height and width we want; so as long as you understand that they will be constrained by other factors (like the size of the parent container), it's safe to think of them in terms of setting the height and width of a view. We've already used `HeightRequest` and `WidthRequest` to specify the size of images in `CellBoutique`, as well as several other places, and we'll look at them again in the next section.

4.4.2 Clipping

Sometimes you need to modify the shape of a view, and this is where the `Clip` property comes in. `Clip` is on the `VisualElement` base class, so can be specified on any control or layout, and is of type `Geometry`.

There are three predefined ‘simple’ geometries that you can use in .NET MAUI which you can assign to the `Clip` property of a view: `EllipseGeometry`, `LineGeometry` and `RectangleGeometry`. The names are self-evident and allow you to specify an ellipse, a line or a rectangle.

Let's add a `Clip` to the `Image` in the `DataTemplate` of our `CellBoutique` app to make each image an ellipse. We'll need to reduce the height and width of the image to allow the full ellipse to be visible. Then we'll specify the `Clip` property with `EllipseGeometry`, and set the `Center` property to the `x` and `y` values that are half the width and height, and do the same for the `RadiusX` and `RadiusY` properties.

Listing 4.2 shows the updated `MainPage.xaml` file in `CellBoutique`. Added or changed lines are shown in **bold**.

Listing 4.2 Adding ellipse clipping to CellBoutique

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

    <CarouselView ItemsSource="{Binding Products}"
        Margin="10">
        <CarouselView.ItemsLayout>
```

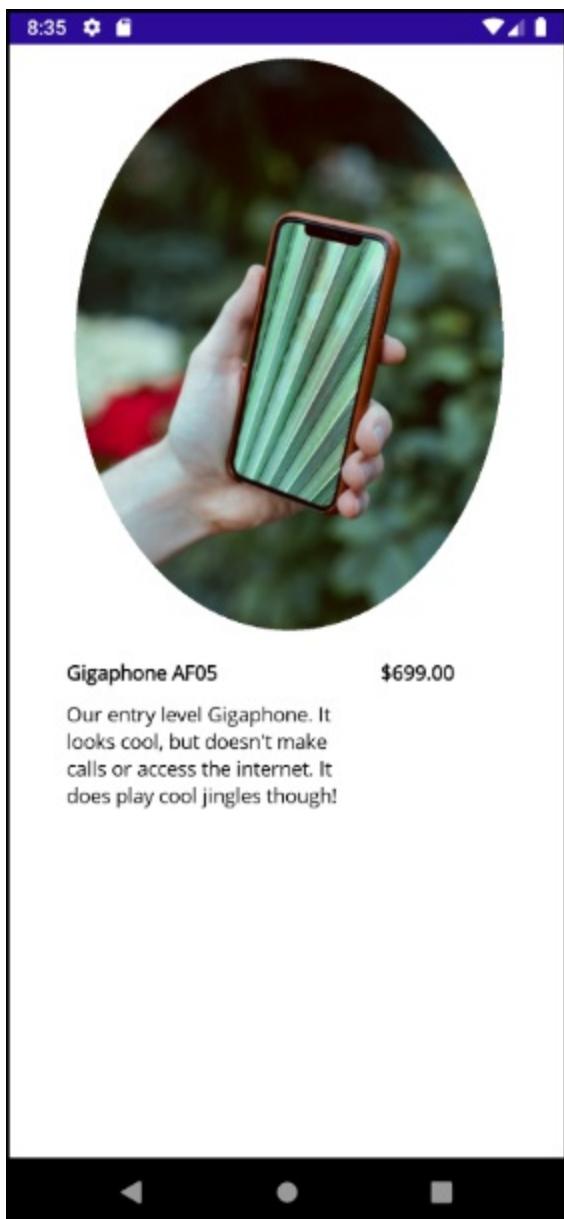
```

        <LinearItemsLayout Orientation="Horizontal"
                           ItemSpacing="60"/>
    </CarouselView.ItemsLayout>
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <VerticalStackLayout Spacing="20">
                <Image Source="{Binding Image}"
                       WidthRequest="300" #A
                       HeightRequest="400" #B
                       Aspect="AspectFill"
                       HorizontalOptions="Center">#C
                    <Image.Clip>#D
                        <EllipseGeometry Center="150, 200" #E
                                         RadiusX="150" #F
                                         RadiusY="200"/> #G
                    </Image.Clip>
                </Image>
                <HorizontalStackLayout Spacing="20">
                    <VerticalStackLayout WidthRequest="200"
                                         Spacing="10">
                        <Label Text="{Binding Title}"
                               FontAttributes="Bold" />
                        <Label Text="{Binding Description}"
                               LineBreakMode="WordWrap"/>
                    </VerticalStackLayout>
                    <Label FontAttributes="Bold"
                           HorizontalTextAlignment="End"
                           Text="{Binding Price, StringFormat='{0'
                                         '#0.00'}/}>
                </HorizontalStackLayout>
            </VerticalStackLayout>
        </DataTemplate>
    </CarouselView.ItemTemplate>
</CarouselView>
</ContentPage>

```

Run CellBoutique now and you should see a result similar to figure 4.9.

Figure 4.9 CellBoutique running on Android. The images have been clipped using EllipseGeometry.



In .NET MAUI, we're not limited to these simple geometries. You can also use a *composite geometry*, which is where you combine more than one geometry to achieve the effect you want. To do this, rather than assign a geometry to the `Clip` property of a view, you can assign a `GeometryGroup`. Inside this group you can specify as many geometries as you like.

Let's say the marketing manager of CellBoutique asks you to update the app for the Christmas period by displaying all the images in a snowman shape. We can use a composite geometry to achieve this by combining two ellipse geometries. Listing 4.3 shows this effect added to CellBoutique.

Listing 4.3 CellBoutique with a composite geometry clipping the images

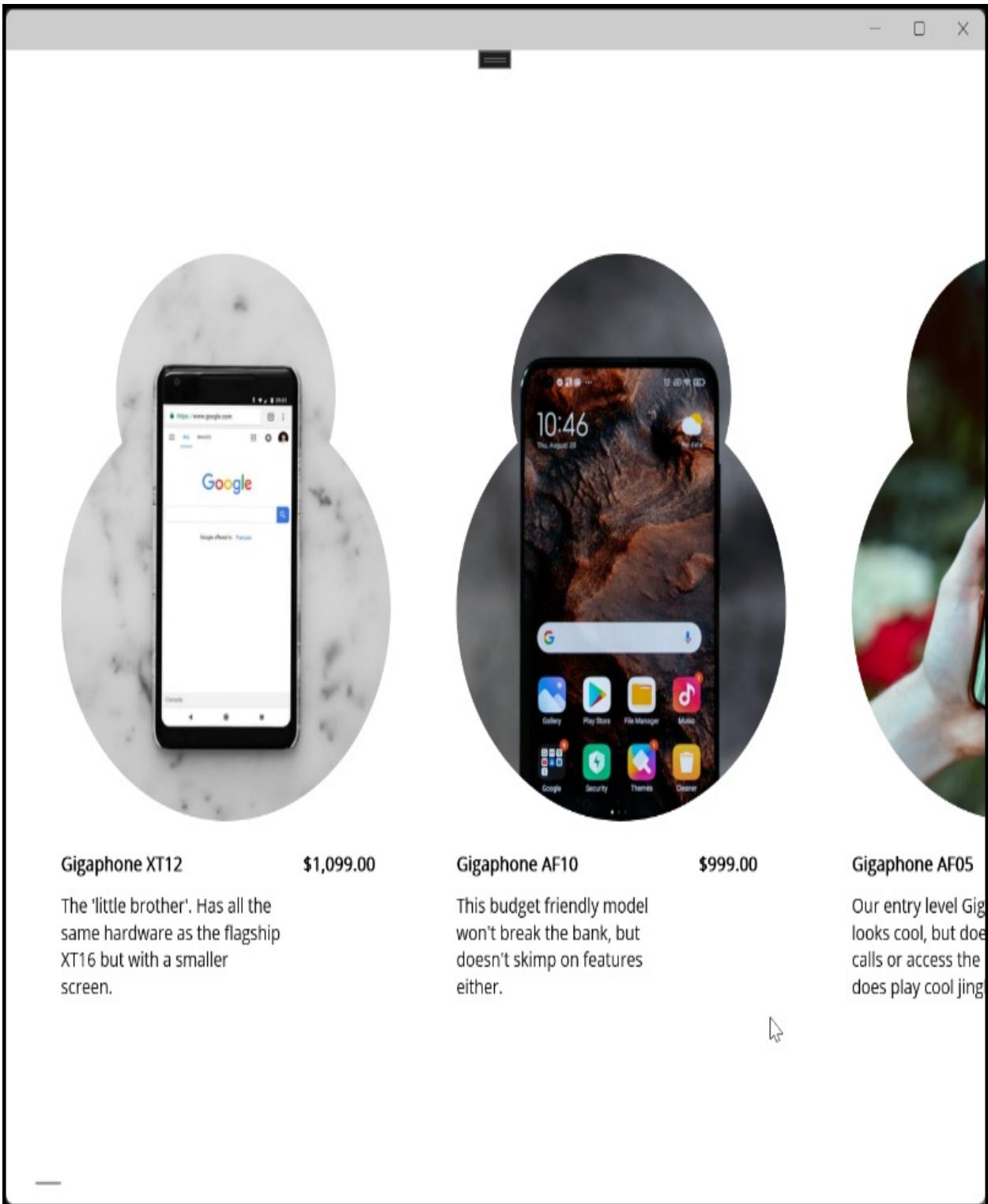
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="Stacks.MainPage">

    <CarouselView ItemsSource="{Binding Products}"
        Margin="10">
        <CarouselView.ItemsLayout>
            <LinearItemsLayout Orientation="Horizontal"
                ItemSpacing="60"/>
        </CarouselView.ItemsLayout>
        <CarouselView.ItemTemplate>
            <DataTemplate>
                <VerticalStackLayout Spacing="20">
                    <Image Source="{Binding Image}"
                        WidthRequest="300"
                        HeightRequest="400"
                        Aspect="AspectFill"
                        HorizontalOptions="Center">
                        <Image.Clip>
                            <GeometryGroup> #A
                                <EllipseGeometry Center="150, 100"
                                    RadiusX="100"#C
                                    RadiusY="100"/>#D
                                <EllipseGeometry Center="150, 250"
                                    RadiusX="150"#F
                                    RadiusY="150"/> #G
                            </GeometryGroup>
                        </Image.Clip>
                    </Image>
                    <HorizontalStackLayout Spacing="20">
                        <VerticalStackLayout WidthRequest="200"
                            Spacing="10">
                            <Label Text="{Binding Title}"
                                FontAttributes="Bold" />
                            <Label Text="{Binding Description}"
                                LineBreakMode="WordWrap"/>
                        </VerticalStackLayout>
                        <Label FontAttributes="Bold"
                            HorizontalTextAlignment="End"
                            Text="{Binding Price, StringFormat='{0'
                                </HorizontalStackLayout>
                            </VerticalStackLayout>
                        </DataTemplate>
                    </CarouselView.ItemTemplate>
```

```
</CarouselView>  
</ContentPage>
```

If you run CellBoutique now, you should see a result similar to figure 4.10.

Figure 4.10 CellBoutique running on Windows. The images have been clipped with a composite geometry comprised of two ellipses: a larger one at the bottom to create the ‘body’ of a snowman shape, and a smaller one at the top to create the ‘head’.



In addition to simple and composite geometries, you can also create any arbitrary shape by using a Path. We're not going to cover Path geometries here, but you can consult the .NET MAUI documentation to learn more about

this topic.

4.4.3 Borders

One of the stated goals of .NET MAUI was “borders everywhere”, and they delivered on this promise. It’s simple to apply a border to any view in .NET MAUI by simply wrapping that control in a border.

You can wrap any view in a `Border` to apply the border to it, by specifying the `Stroke`, `StrokeThickness`, and `StrokeShape` properties. The `Stroke` property is a `Brush`, which means you can define a linear gradient, a radial gradient, or a single colour, which you can specify using the `Colors` enum, a hex value, or a static or dynamic resource. `StrokeThickness`, as the name suggests, defines how thick the border will be in DIUs.

Brushes

There are a few places in .NET MAUI where you can use a `Brush`, including the `Stroke` property of a `Border`, but also for view backgrounds.

There are three types of `Brush`: `SolidColorBrush`, `LinearGradientBrush` and `RadialGradientBrush`. `SolidColorBrush` is the default, which is why you can specify a single colour using one of the available methods (enum, hex, or by referencing a resource) and is what we will use in this section, but you can use a gradient for a `Border` too.

We will learn more about `LinearGradientBrush` in chapter 7, where we will use it to set a gradient for a page background.

`StrokeShape` defines the shape of the border around the view. You can use `Ellipse`, `Rectangle`, or `RoundRectangle` (a rectangle with rounded corners) pre-defined borders, but you’re not limited to rectangles and ellipses. You can draw more complex shapes using the `Path`, `Polygon` and `Polyline` implementations of the `IShape` interface (the type that `StrokeShape` expects) too, or even just a simple line.

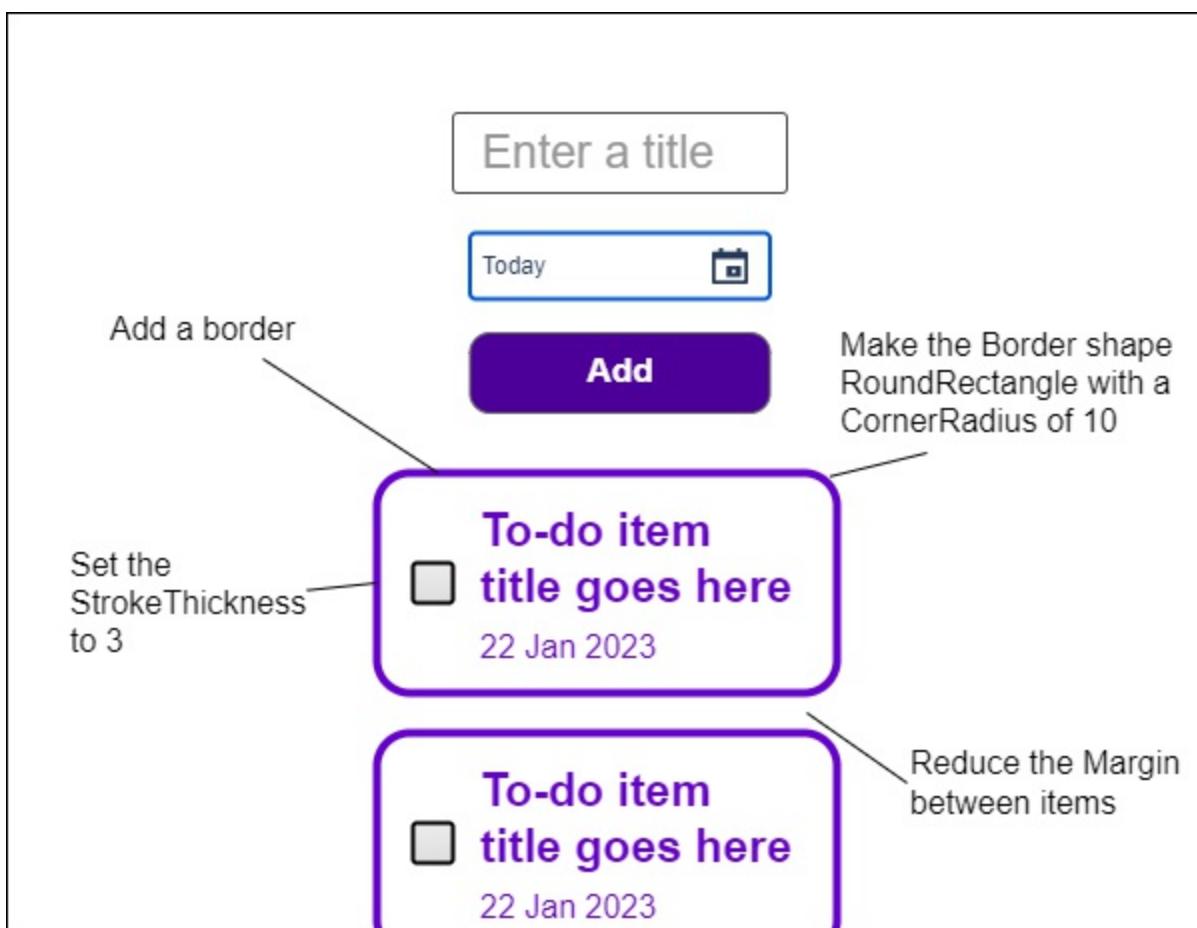
Let’s add borders to `MauiTodo`. So far, we’ve been relying on layout alone to visually distinguish the different to-do items in the `CollectionView`, but let’s

add borders to them now to turn the individual items into cards.

We'll add a `RoundRectangle` border to add a card like effect to the to-do items. Open `MainPage.xaml` and wrap the `Grid` inside the `DataTemplate` in a `Border` tag. Set the `Stroke` property to `Black` and the `StrokeThickness` to `3`.

As we'll be using a card style to display the to-do items, we can also reduce the spacing as we won't need as much to separate individual items in the list.

Figure 4.11 Adding borders to the to-do items in MauiTodo. The Border Shape will be RoundRectangle with a CornerRadius of 10. The Border Stroke will be the Primary colour StaticResource, and the StrokeThickness will be 3. Using Borders also means we can reduce the space between items in the collection.



Listing 4.4 shows the updated `DataTemplate` in the `MainPage.xaml` file in MauiTodo, with the main changes shown in **bold**. Update your code to match this listing.

Listing 4.4 Borders in MauiTodo

```
<DataTemplate>
    <Border Stroke="{StaticResource PrimaryColor}"#A
            StrokeThickness="3"#B
            Padding="5"#C
            Margin="0, 10"> #D
        <Border.StrokeShape>#E
            <RoundRectangle CornerRadius="10"/>#F
        </Border.StrokeShape>
        <Grid WidthRequest="325"
              ColumnDefinitions="1*, 5*"
              RowDefinitions="Auto, 25"
              x:Name="TodoItem">

            <CheckBox VerticalOptions="Center"
                      HorizontalOptions="Center"
                      Grid.Column="0"
                      Grid.Row="0" />

            <Label Text="{Binding Title}"
                  FontAttributes="Bold"
                  LineBreakMode="WordWrap"
                  HorizontalOptions="StartAndExpand"
                  FontSize="Medium"
                  Grid.Row="0"
                  Grid.Column="1"/>

            <Label Text="{Binding Due, StringFormat='{0:dd MMM yy}"
                  VerticalOptions="End"
                  Grid.Column="1"
                  Grid.Row="1"/>
        </Grid>
    </Border>
</DataTemplate>
```

Note also that not all four corners need to have the same radius. You can add four individual values here separated by a space, corresponding to the top left, top right, bottom right, and bottom left corners respectively.

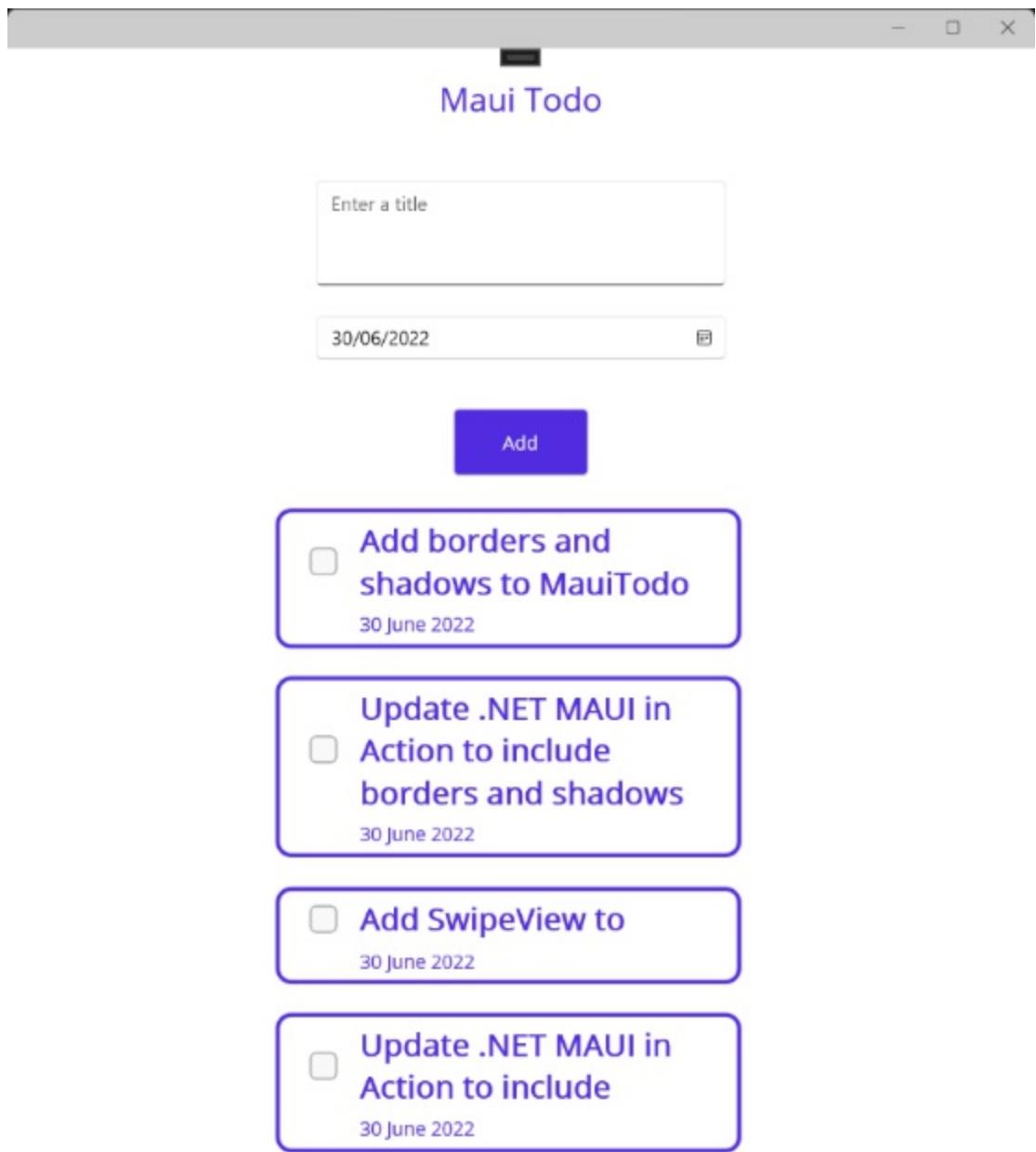
There's also a nice shorthand way to define the `StrokeShape` though. We can specify the property within the `Border` tag directly. Listing 4.5 shows this shorthand approach.

Listing 4.5 Simplified StrokeShape

```
<Border Stroke="Black"  
       StrokeThickness="3"  
       StrokeShape="RoundRectangle 10" #A  
       Padding="10">  
  <Grid>  
    ...  
  </Grid>  
</Border>
```

Run the MauiTodo app now and you should see it with the borders added. Figure 4.12 shows MauiTodo with the borders and reduced spacing running on Windows.

Figure 4.12 MauiTodo running on Windows. A Border has been added to each to-do item, which also lets us reduce the space between items which is necessary without the border.



4.4.4 Shadows

.NET MAUI makes it easy to add a shadow to any view. The `Shadow` property is on the `VisualElement` base class, which every control and layout inherits.

Shadow requires four properties:

- Brush, which specifies the colour of the shadow and can be set with the Colors enum
- Opacity, which specifies how opaque the shadow will be and can be specified with a value between 0 and 1
- Radius, which defines the radius of the shadow and can be specified with a number in DIUs
- Offset, which specifies the offset of the shadow (so conceptually the position of the light source). The last value, Offset, is of type Point, so requires a pair of values (also in DIUs) for the x and y coordinates.

As mentioned, you can add a Shadow to any view. In the MauiTodo app, we'll add a Shadow to the Border, as this is the outermost view of any card, and so the view that we want to apply the Shadow to. Listing 4.6 shows a Shadow added to the DataTemplate in the MauiTodo app, with the added Shadow in **bold**.

Listing 4.6 Border with Shadow

```
<Border Stroke="{StaticResource Primary}"  
       StrokeThickness="3"  
       StrokeShape="RoundRectangle 10"  
       Padding="10">  
    <Border.Shadow>#A  
        <Shadow Brush="Black"#B  
              Offset="20, 20"#C  
              Radius="40"#D  
              Opacity="0.8" />#E  
    </Border.Shadow>  
    <Grid ...>  
        ...  
    </Grid>  
</Border>
```

If you run MauiTodo now, you should see a result similar to figure 4.13.

Figure 4.13 MauiTodo running on Android, with Shadow added to the Border



4.4.5 Gesture Recognisers

Capacitive touchscreens became ubiquitous following the launch of the first iPhone, and along with them, touch gestures have also become ubiquitous as a UI interaction paradigm. .NET MAUI supports the following five gesture recognisers:

- Tap
- Pan
- Swipe
- Pinch
- Drag and drop

With the exception of pinch, these gestures have analogous interactions that can be achieved using a mouse or equivalent cursor device.

The `View` base class has a collection called `GestureRecognizers`, and as this is inherited by all controls and layouts, these gestures can be added to any view in .NET MAUI. We'll see some of these in action later in the book.

4.4.6 RefreshView

Pull to refresh is another ubiquitous UI paradigm that's become popular since the proliferation of capacitive touchscreens. You pull down from the top of the screen and then let go to update the content on the screen.

.NET MAUI makes it easy to implement this with `RefreshView`.

`RefreshView` is another wrapper that can be placed around any other view to add the pull to refresh functionality, including scrolling views like `ScrollView` or the list and collection views we looked at earlier in the chapter. It lets you scroll to the top before invoking the refresh functionality which prevents it from interfering with scrolling views.

`RefreshView` has two bindable properties you can use: `Command` and `IsRefreshing`. The `Command` is the logic to be executed to refresh the view, and `IsRefreshing` is Boolean that you can set to false when the logic has been completed. When the user invokes the refresh, an `ActivityIndicator`, built into the `RefreshView` rather than one you explicitly define yourself, will be displayed until `IsRefreshing` is set to false.

4.4.7 SwipeView

I've said a few times that lists and collections are at the heart of most apps. Tapping (or clicking) on items in a list or collection is an established

interaction paradigm for selecting or interacting with that item. Another UI paradigm that has become ubiquitous is the use of a swipe to reveal additional functionality. For example, in the Outlook mobile app, you can swipe on an item in your mailbox to reveal configurable actions such as delete or archive.

In .NET MAUI we can use `SwipeView` to add this functionality to any view. Like `Border`, `SwipeView` wraps other views, which is how you add the `SwipeView` functionality to them.

`SwipeView` has four collections of `SwipeItems` called `LeftItems`, `RightItems`, `TopItems` and `BottomItems`. You can add `SwipeItems` to these collections which will be revealed depending on which direction a user swipes. A property called `Mode` with two options, `Execute` and `Reveal`, is used to determine what happens when your user swipes on the `SwipeView`.

Let's add `SwipeView` to `MauiTodo`. We'll add a `SwipeItem` to the `LeftItems` collection to delete the to-do item, and a `SwipeItem` to `RightItems` that marks it as done (something that we can't currently do with the checkbox and won't be able to until we get to the MVVM pattern in chapter 7).

Taking inspiration from fruit, we'll make the delete `SwipeItem` tomato red, and the done `SwipeItem` lime green. We'll also add an icon for each of these (you can find the `check.svg` and `delete.svg` files in this chapter's online resources).

Listing 4.7 shows the updated `DataTemplate` for `MauiTodo` with the `SwipeView` included.

Listing 4.7 Adding SwipeView

```
<DataTemplate>
    <SwipeView>#A
        <SwipeView.LeftItems>#B
            <SwipeItems Mode="Execute">#C
                <SwipeItem Text="Delete">#D
                    IconImageSource="delete"#E
                    BackgroundColor="Tomato"/>#F
            </SwipeItems>
        </SwipeView.LeftItems>
```

```
<SwipeView.RightItems>#G
    <SwipeItems Mode="Execute">#H
        <SwipeItem Text="Done"#I
            IconImageSource="check"#J
            BackgroundColor="LimeGreen"/>#K
    </SwipeItems>
</SwipeView.RightItems>

<Border ...>
    <Border.Shadow>
    ...
        </Border.Shadow>
    <Grid ...>
        ...
    </Grid>
</Border>
</SwipeView>
</DataTemplate>
```

Run the MauiTodo app now. Add some to-do items if you haven't already and try swiping left and right on them. You should see something similar to figure 4.14.

Figure 4.14 MauiTodo with SwipeView added. The SwipeItemsMode is set to Execute, which means that as the user starts to swipe, the action is revealed, and completing the swipe gesture executes the action.

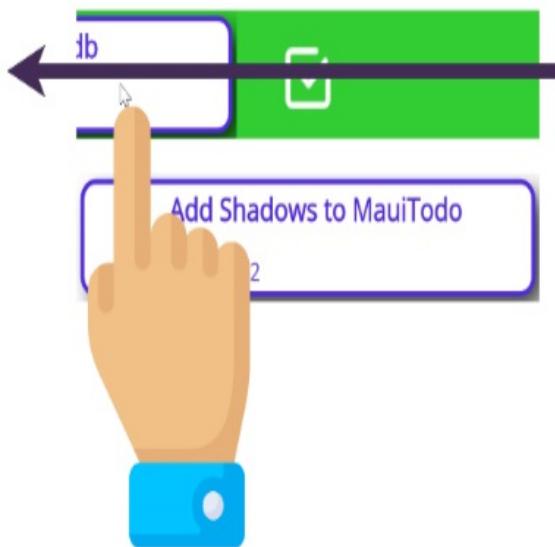


Maui Todo

Enter a title

7/2/2022

Add



Maui Todo

Enter a title

7/2/2022

Add



Swiping on the cards starts to reveal the action that the swipe provides and completing the swipe gesture (swiping the item completely to the left or the right) will execute the action. The alternative Mode we can use is Reveal, which instead of executing shows the available actions to the user, who must

then explicitly execute them with a tap or click. **Change the SwipeItems Mode in MauiTodo from Execute to Reveal.** The result should be like figure 4.15.

Figure 4.15 MauiTodo with SwipeView added. The SwipeItems mode is set to Reveal, which means that when a user swipes, additional actions are revealed which the user must then execute with a tap or a click.



Maui Todo

Enter a title

7/2/2022

Add

date MauiTodo db

02 Jul 2022



Add Shadows to MauiTodo

02 Jul 2022



Maui Todo

Enter a title

7/2/2022

Add

Update MauiTodo db

02 Jul 2022

Add Shadows to MauiTodo

02 Jul 2022



You can choose whether to use Execute or Reveal depending on your needs. Reveal is a better option when you want to show multiple actions.

To execute the actions that your SwipeView adds, you can use an event handler called `Invoked`, or alternatively you can use the `Command` and `CommandParameter` bindable properties. Let's add an event handler to these. Add the method from listing 4.8 to `MainPage.xaml.cs`.

Listing 4.8 the SwipeItem_Invoked method

```
private async void SwipeItem_Invoked(object sender, EventArgs e)
{
    var item = sender as SwipeItem; #B

    await App.Current.MainPage.DisplayAlert(item.Text, $
}
```

Next, let's wire up this event handler to the swipe items. Listing 4.9 shows the `DataTemplate` with the `Invoked` event handler added to the swipe items. Add the lines in **bold**.

Listing 4.9 The DataTemplate with the Invoked event handler

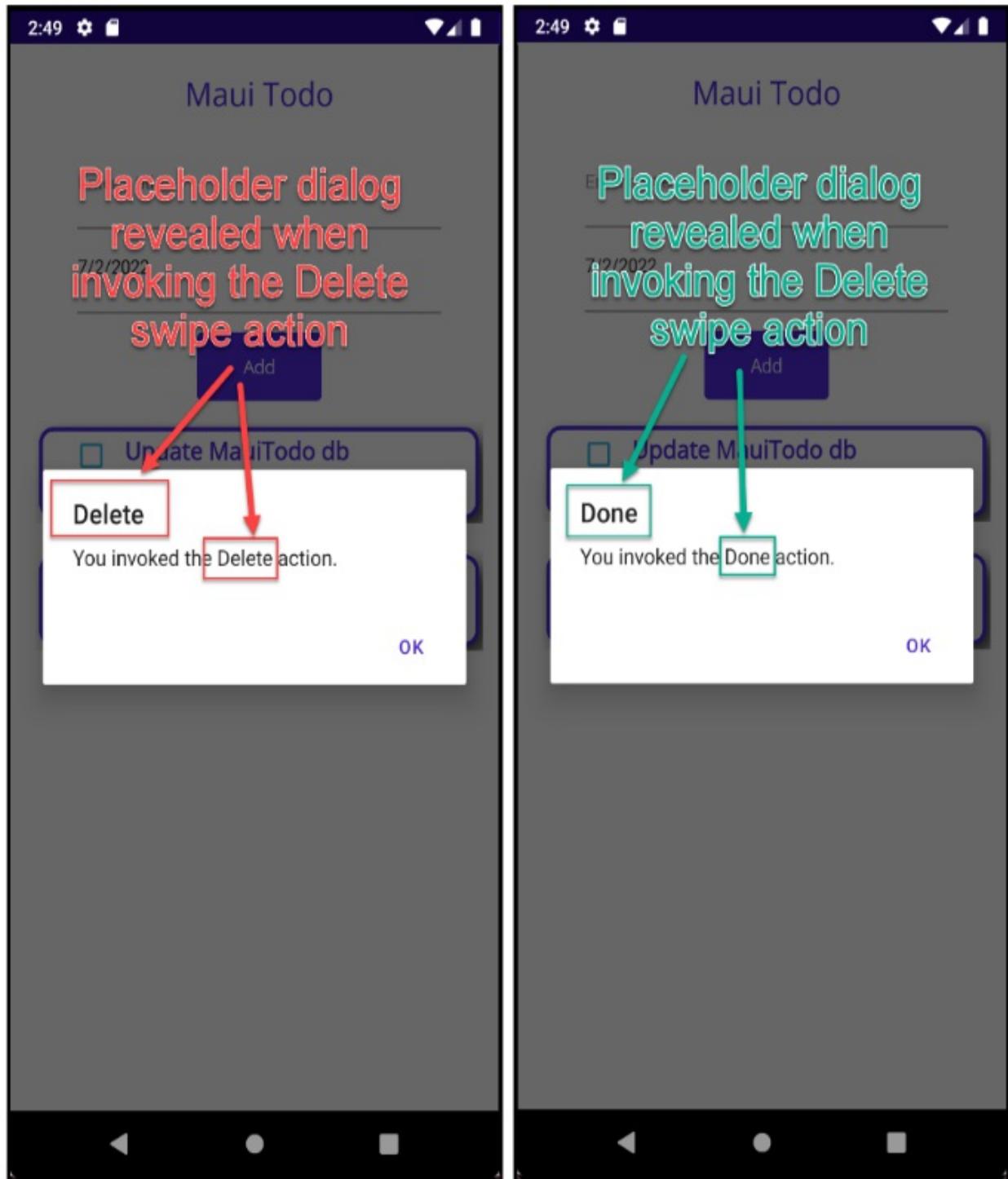
```
<DataTemplate>
    <SwipeView>
        <SwipeView.LeftItems>
            <SwipeItems Mode="Reveal">
                <SwipeItem Text="Delete"
                    IconImageSource="delete"
                    BackgroundColor="Tomato"
                    Invoked="SwipeItem_Invoked"/>#A
            </SwipeItems>
        </SwipeView.LeftItems>

        <SwipeView.RightItems>
            <SwipeItems Mode="Reveal">
                <SwipeItem Text="Done"
                    IconImageSource="check"
                    BackgroundColor="LimeGreen"
                    Invoked="SwipeItem_Invoked"/>#B
            </SwipeItems>
        </SwipeView.RightItems>
        <Border ...>
            <Border.Shadow>
                ...
            </Border.Shadow>
        <Grid ...>
```

```
    ...
  </Grid>
</Border>
</SwipeView>
</DataTemplate>
```

If you run the app now and execute the swipe items (by sliding the item to reveal the action, then tapping the icon), you should see an alert. The result is shown in figure 4.16.

Figure 4.16 MauiTodo running on Android. Executing a swipe action, by swiping one of the to-do items to the side and then tapping on the icon that is revealed, calls an event handler, which parses the sender (in this case a SwipeItem) and displays an alert using the Text property of the sender.



These alerts are placeholders for the functionality to mark a to-do item as done or to delete one. We can't actually implement this functionality just yet; later, in chapter 9 we'll see why this is the case, and how we can refactor MauiTodo to use the MVVM pattern, and how that will help us solve this

problem.

4.5 Summary

- Views are the visual components of an app. They include pages, which are navigable sections of an app, layouts, which control how things are arranged on screen, and controls, which show something to or get something from the user.
- You can use a number of built-in controls in .NET MAUI to display information to a user or accept input from them.
- .NET MAUI provides a number of controls that can bind to a data source and use a template to render each item in the data source to the screen.
- You can modify the appearance of any control or layout to include borders and shadows and can clip the shape of a layout or control too using a geometry.
- You can add recognisers for gestures, such as tap and swipe, to views in .NET MAUI to add well-known UX paradigms to your apps.

5 Layouts

This chapter covers:

- The different types of built-in layout you can use in .NET MAUI apps
- The importance of layouts and when to use which one
- How and when to use ScrollView
- How to combine layouts to create a UI

Layout is the most important aspect of a UI design. UI design consists fundamentally of three aspects: layout, typography, and color (of course there are more, but at a high level these are the key components of a design). You can create a great looking UI in black and white with only one typeface, but fancy typography and pretty colors won't rescue a bad layout. From the experts:

Layout is the most critical piece of website's design. It is quite literally the foundation for the rest of the pieces that will eventually be added to the website.

Design for Developers, Stephanie Stimac, Manning 2022

Stimac uses the word 'website' here, but she's really talking about UI. And that can be web, desktop, or mobile UI. As a .NET MAUI developer you're in luck, as we have plenty of options for how we can lay out our UI. The layouts included in .NET MAUI are:

- `HorizontalStackLayout`
- `VerticalStackLayout`
- `Grid`
- `FlexLayout`
- `BindableLayout`
- `AbsoluteLayout`

We'll look at the first three (`Grid`, `HorizontalStackLayout`, and

`VerticalStackLayout`), with examples of varying complexity, and then look at how you can combine them to achieve any design. We'll look at the last three (`FlexLayout`, `BindableLayout` and `AbsoluteLayout`) in the next chapter.

RelativeLayout – where has it gone?

If you've previously used Xamarin.Forms, you may be familiar with another layout called `RelativeLayout`. While `RelativeLayout` is included in .NET MAUI for compatibility with Xamarin.Forms, you shouldn't use it in .NET MAUI apps because it can be expensive to calculate (meaning it takes more CPU cycles), resulting in slower apps.

You can achieve the same results as a `RelativeLayout` by using a `Grid`. If you're migrating from Xamarin.Forms, consider updating your layout to use a `Grid` instead. Of course, with `RelativeLayout` included for compatibility, you have the luxury of making this update after you upgrade. See Appendix B for more details.

`RelativeLayout` is not completely gone though. While we won't use it as a layout, we will use it for layout constraints, which we'll cover in the chapter on Advanced Layout Concepts.

`ScrollView` is also included in this chapter, even though it's technically a control and not a layout. The reason for this is that you will usually use it as part of how you lay out your UI, rather than to show something to the user or get feedback from them. That, combined with the fact that it displays child controls, means it essentially behaves as a layout.

By the end of this chapter, you'll understand why `Grid` is the most important layout and why it will form the basis of any non-trivial UI.

5.1 Grid

`Grid` is the most powerful layout in .NET MAUI, and you will likely find that most of your UI layouts are implemented with a `Grid`. As the name suggests, `Grid` lets you arrange child elements in rows and columns. It's easy to picture

a table when you hear ‘rows’ and ‘columns’, but this would be a limiting way to think about the `Grid` layout in .NET MAUI.

A `Grid` isn’t used for *displaying* rows and columns; there are better options in .NET MAUI for displaying tabular data. Instead, a `Grid` uses rows and columns to arrange your views on the screen. If you’re familiar with the popular design tool Figma, the concept is the same as layout grids that Figma uses, as well as most other design tools.

In this section, we’ll look first at a simple example of using `Grid`. We’ll then see a more complex design and gain an appreciation for using `Grid` to build awesome layouts. At the end of the chapter, we’ll see another practical, hands-on example that uses `Grid` as the chief layout for a page, in tandem with other layouts to replicate the UI of a well-known app.

5.1.1 Grid Basics

You might think that using a `Grid` means following squares like on the graph paper you used in school. This isn’t how it works in .NET MAUI. To use a `Grid` in .NET MAUI, you decide how many rows and columns your view will be laid out in, and what the sizes of those rows and columns will be.

If you’re familiar with the classic “three column layout” in web design, you should already be familiar with the concept of a grid system. It simply means that you lay out your UI in rows and columns, rather than use a fixed size grid to construct your UI.

Let’s start with a simple example, and one that logically lends itself to a simple grid pattern: a calculator. Let’s start right from the very beginning, with a sketch of what a calculator app might look like.

Figure 5.1 A quick sketch of the design for our calculator app. This design is comprised of five rows and four columns

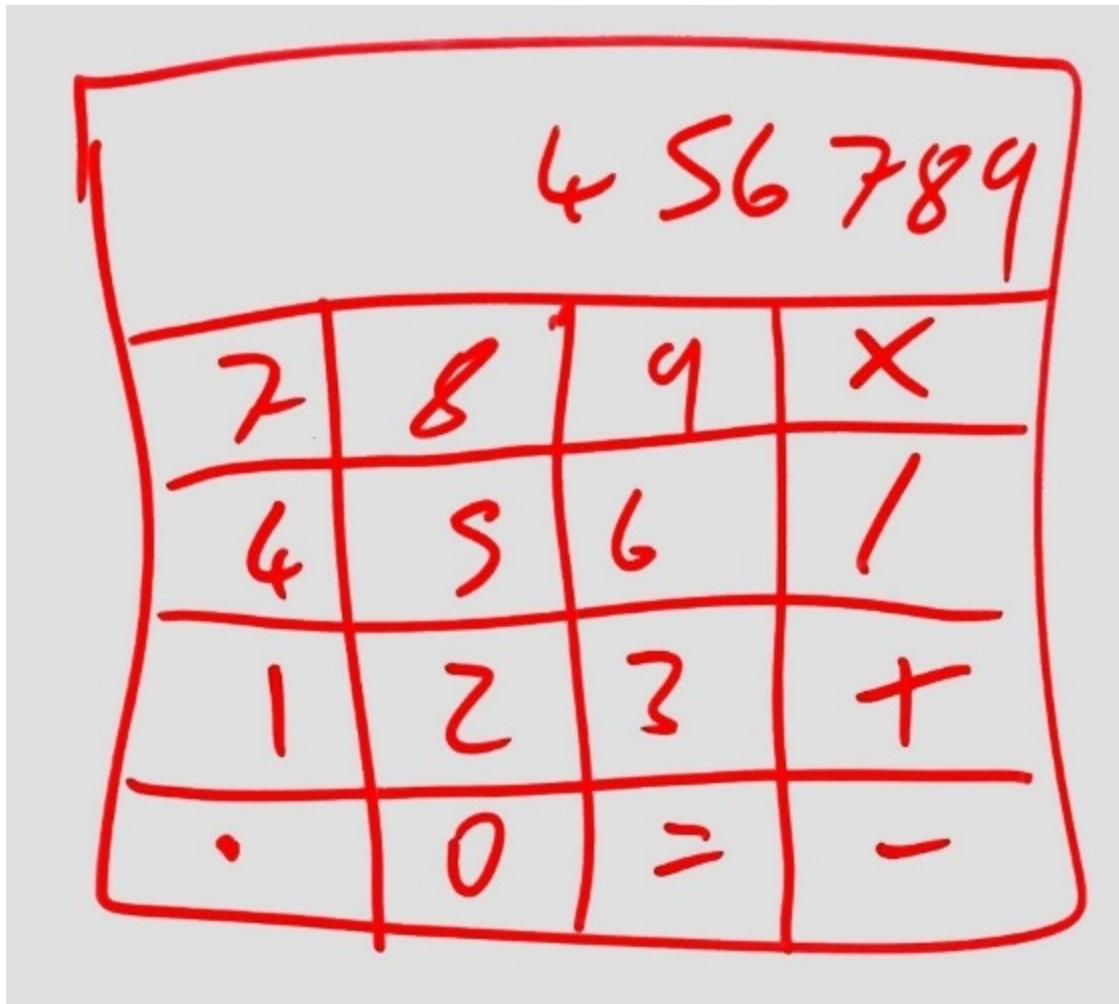


Figure 5.1 shows a simple whiteboard sketch of our calculator app's UI. We can see that there is a 'screen' at the top to display the numbers the user is entering, and the result of their calculation, and then rows of buttons for entering numbers and selecting mathematical operators.

If we think of this design as a grid, we can easily say that it has five rows and four columns. Row and column numbers start at 0, so we would define the screen as being in row 0. The buttons would all be in rows 1 to 4 and columns 0 to 3.

Let's build this calculator app in .NET MAUI.

5.1.2 Building MauiCalc

Create a new blank .NET MAUI app called MauiCalc. Open the

`MainPage.xaml` file and delete everything between the `<ContentPage>...` `</ContentPage>` tags. We know that this layout is going to be a Grid, so add a Grid to the page:

Listing 5.1 The MauiCalc MainPage with the grid added

```
<?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="MauiCalc.MainPage">

    <Grid> #A
    </Grid>
</ContentPage>
```

Now that we've added the Grid, we need to add row and column definitions. Row definitions have a Height property and column definitions have a Width property. There are two ways to define rows and columns in a Grid. The first is to add the definitions as XAML elements inside the Grid:

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

In this example, the Grid has two rows and two columns, and instead of specifying a width or height, we've used an asterisk (*), indicating that we want these to be proportionally sized. We'll talk about row and column sizing in the next section, but for now know that proportionally sized means they will be equally distributed. There's a much simpler way though, and that is to directly specify the row and column definitions as part of the Grid declaration.

```
<Grid RowDefinitions="*, *"
    ColumnDefinitions="*, *">
</Grid>
```

This gives us the same result – a Grid with two rows and two columns. But you can see that declaring the row and column definitions is much easier simpler this way.

We can also specify row and column spacing, using the RowSpacing and ColumnSpacing properties respectively. By default, there will be no space at all between columns and rows in a Grid but using these we can specify that we want there to be a gap between each row and column, specified in DIUs.

Let's add these row and column definitions to MauiCalc now. Listing 5.2 shows the `MainPage.xaml` code with the row and column definitions added. The added parts are in **bold**.

Listing 5.2 Row and Column definitions inline

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

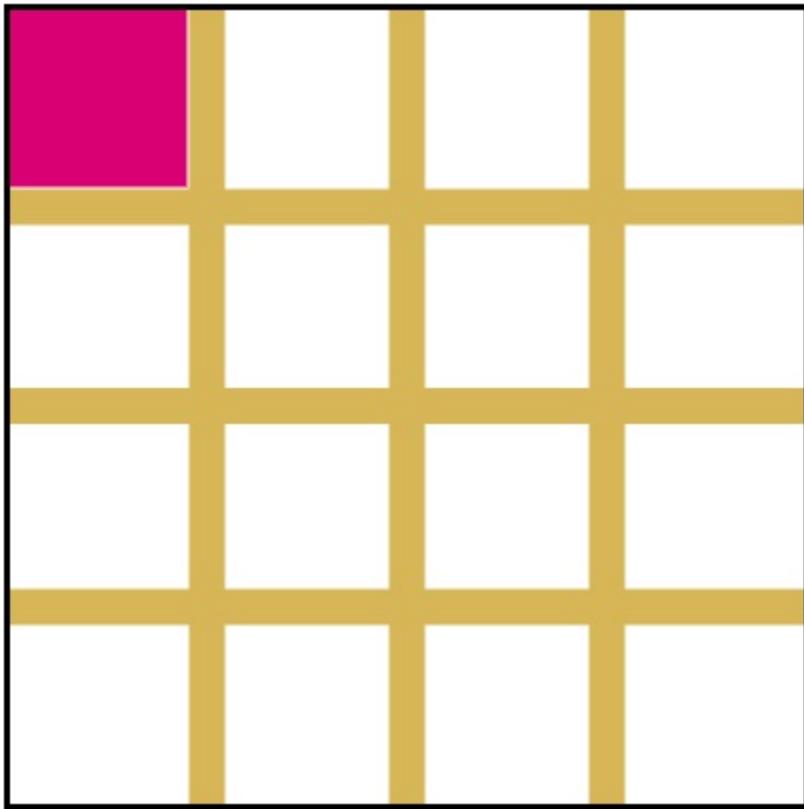
    <Grid ColumnDefinitions="*, *, *, *"#A
          RowDefinitions="*, *, *, *"#B
          RowSpacing="2"#C
          ColumnSpacing="2">#D
    </Grid>
</ContentPage>
```

MauiCalc has a simple UI, and this Grid declaration is all we need to define its layout. Now we need to start adding some controls.

To place controls in a Grid, we can use attached properties called `Grid.Row` and `Grid.Column`. These can be used in any control or layout to position them where we want them in a Grid. Grid rows and columns start at 0 which for a row is at the topmost of the Grid, and for a column is leftmost.

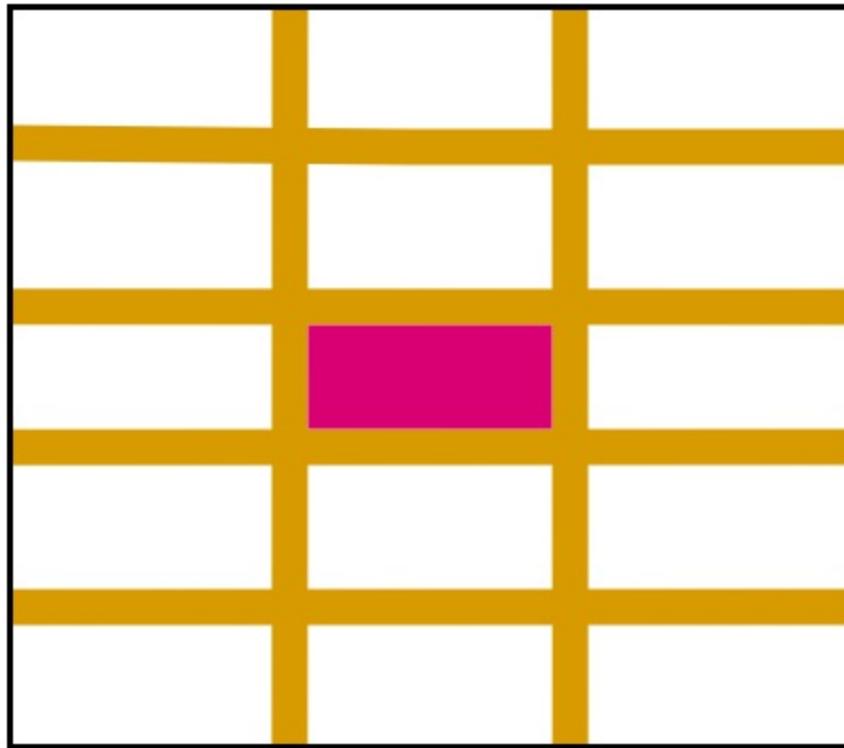
So, if we want something in the upper left corner, we will place it in `Grid.Row 0` and `Grid.Column 0`.

Figure 5.2 A Grid layout with four rows and four columns, and a view positioned in Grid.Row 0, Grid.Column 0



If we have, say, a Grid with five rows and three columns, and we want to place our view in the middle, we will place it in Grid.Row 2 and Grid.Column 1.

Figure 5.3 A Grid layout with five rows and three columns, with a View positioned in Grid.Row 2, Grid.Column 1.

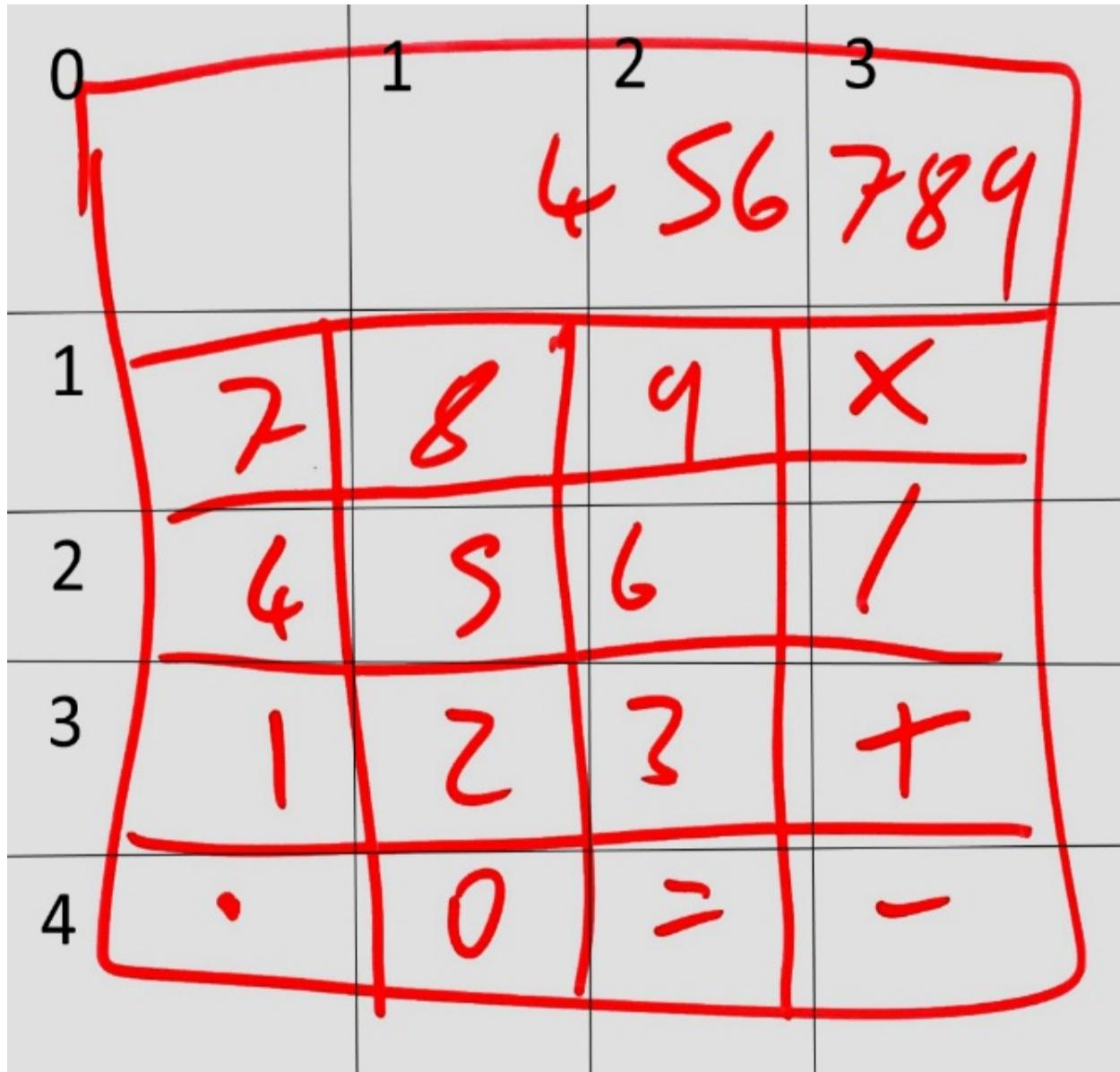


You can use these attached properties to position any view at the desired location within a `Grid`:

```
<Button Grid.Row="2"  
       Grid.Column="1"  
       Text="Click me!" />
```

Let's add the buttons now to our calculator app. Figuring out which row and column each button needs to go into is simple enough, but to make it easier, let's look at the design again with a numbered grid overlaid, as in figure 5.4.

Figure 5.4 The MauiCalc design with a numbered grid added. This helps us easily determine which row and column to assign each button to.



Looking at this we can easily see that the button for the number 1, for example, will be in row 3, column 0, and the button for the number 9 will be in row 1, column 2. None of the buttons are in row 0 as we have reserved this for the ‘screen’.

Listing 5.3 shows the updated `MainPage.xaml` class, with the added buttons shown in **bold**.

Listing 5.3 MainPage with the buttons added

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

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

<Grid ColumnDefinitions="*, *, *, *"
      RowDefinitions="*, *, *, *, *"
      RowSpacing="2"
      ColumnSpacing="2">

    <!-- Row 1 -->

    <Button Grid.Row="1">#A
        Grid.Column="0">#B
        CornerRadius="0">#C
        Text="7"#D
        Clicked="Button_Clicked"/>#E

    <Button Grid.Row="1">#F
        Grid.Column="1">#G
        CornerRadius="0"
        Text="8"
        Clicked="Button_Clicked"/>

    <Button Grid.Row="1"
        Grid.Column="2"
        CornerRadius="0"
        Text="9"
        Clicked="Button_Clicked"/>

    <!-- Row 2 -->

    <Button Grid.Row="2">#H
        Grid.Column="0"
        CornerRadius="0"
        Text="4"
        Clicked="Button_Clicked"/>

    <Button Grid.Row="2"
        Grid.Column="1"
        CornerRadius="0"
        Text="5"
        Clicked="Button_Clicked"/>

    <Button Grid.Row="2"
        Grid.Column="2"
        CornerRadius="0"
        Text="6"
        Clicked="Button_Clicked"/>
```

```
        Clicked="Button_Clicked"/>

    <!-- Row 3 -->
<Button Grid.Row="3">#I
Grid.Column="0"
CornerRadius="0"
    Text="1"
    Clicked="Button_Clicked"/>

<Button Grid.Row="3"
    Grid.Column="1"
    CornerRadius="0"
    Text="2"
    Clicked="Button_Clicked"/>

<Button Grid.Row="3"
    Grid.Column="2"
    CornerRadius="0"
    Text="3"
    Clicked="Button_Clicked"/>

<!-- Row 4 -->

<Button Grid.Row="4">#J
    Grid.Column="0"
    CornerRadius="0"
    Text="."
    Clicked="Button_Clicked"/>

<Button Grid.Row="4"
    Grid.Column="1"
    CornerRadius="0"
    Text="0"
    Clicked="Button_Clicked"/>

<Button Grid.Row="4"
    Grid.Column="2"
    CornerRadius="0"
    Text ="="
    Clicked="Button_Clicked"/>

<!-- Column 3 (Operator buttons) -->

<Button Grid.Row="1"
    Grid.Column="3">#K
    CornerRadius="0"
    Text="+"
```

```

        Clicked="Button_Clicked"/>

    <Button Grid.Row="2"
            Grid.Column="3"
            CornerRadius="0"
            Text="-"
            Clicked="Button_Clicked"/>

    <Button Grid.Row="3"
            Grid.Column="3"
            CornerRadius="0"
            Text="X"
            Clicked="Button_Clicked"/>

    <Button Grid.Row="4"
            Grid.Column="3"
            CornerRadius="0"
            Text="/"
            Clicked="Button_Clicked"/>
</Grid>
</ContentPage>
```

That adds all the Buttons to our calculator app, and there are a couple of things you may have noticed. The first is that each Button has its Clicked event delegated to an event handler called `Button_Clicked`. We'll add a single method to handle *all* button clicks, as adding an event handler for each individual button would be unwieldy. We'll add this method shortly.

The second is that we have started placing Buttons on row 1, rather than row 0 which is the top row. This is because we have left the top row free to place the screen, which will show the numbers the user is entering and the result of any calculations.

We want the screen to take up the whole of the top row, rather than just one column in the row, and to do this we will use a property called `ColumnSpan`. `ColumnSpan` lets you declare that, while your view originates in a particular column, it should span across the specified number of columns. Our calculator app has four columns, and we want the screen to span across all of them. Therefore, we would declare that it be placed in `Grid.Column 0` and have a `ColumnSpan` of four.

You can use the same trick for rows as well as columns. If you want a view

to occupy more than one vertical row, you can use the `RowSpan` property to specify how many rows the view should span across.

Let's add the screen to MauiCalc. Listing 5.4 shows the code to add to `MainPage.xaml`, with the added code shown in **bold**.

Listing 5.4 The Button_Clicked method

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

    <Grid ColumnDefinitions="*, *, *, *"
        RowDefinitions="*, *, *, *"
        RowSpacing="2"
        ColumnSpacing="2">

        <Label Grid.Row="0">#A
            Grid.Column="0"#B
            Grid.ColumnSpan="4"#C
            FontSize="72"#D
            Padding="20"#E
            TextColor="Black"#F
            HorizontalTextAlignment="End"#G
            x:Name="LCD"/>#H

        <!--Buttons omitted for brevity -->

    </Grid>
</ContentPage>
```

Now that we've added the LCD screen, this completes the layout for the calculator app. You can't run it just yet, as we are referencing an event handler that doesn't exist. Add an empty event handler for now just so that we can run the app and review the layout. Listing 5.5 shows the code to add to `MainPage.xaml.cs`.

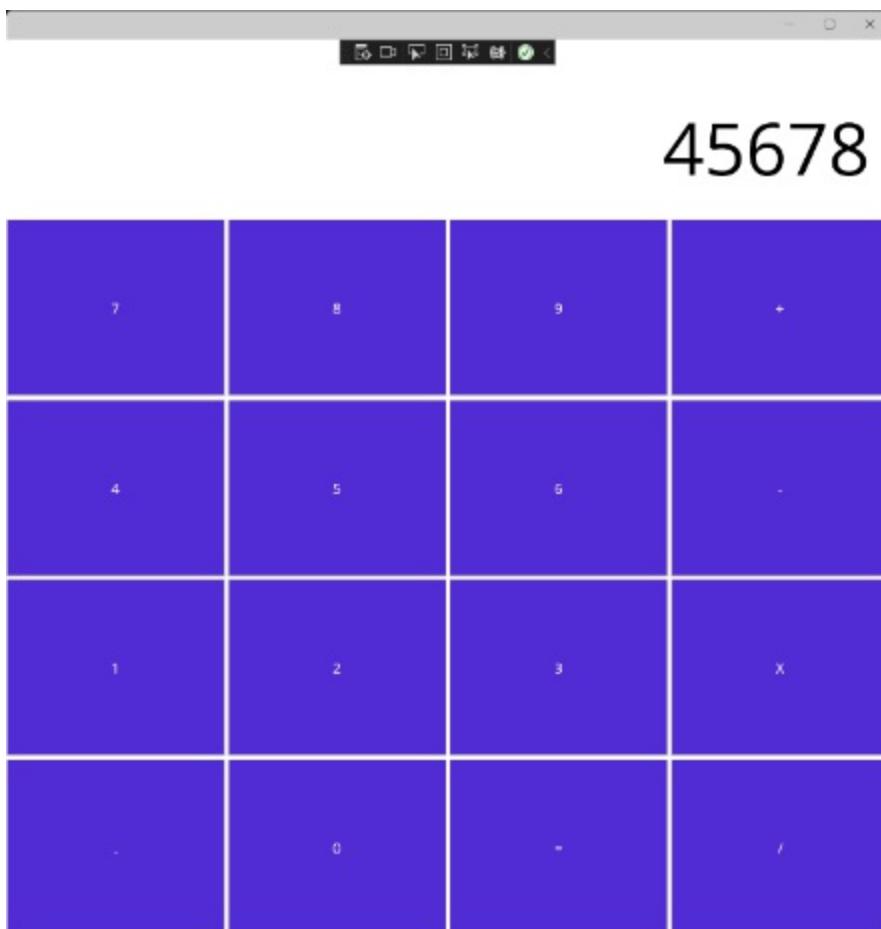
Listing 5.5 the Button_Clicked method.

```
private void Button_Clicked(object sender, EventArgs e)#A
{
```

```
}
```

Now that you've added the method the app will compile and run. **Before you run the app, add some numbers to the Text property of the LCD Label.** This will let you get a feel for what the app will look like. If you run it now, you should see something like figure 5.5.

Figure 5.5 MauiCalc running on Windows, with the Buttons laid out in a Grid, in rows 1 to 4 and columns 1 to 4, and a number display in row 0, spread across all four columns.



Now that we've got our layout sorted, we can start thinking about the other two aspects of this UI: color and typography. We can make some small UI tweaks to give it a more calculator like appearance. First, we'll give the 'screen' a more LCD like background color. We can also use an LCD styled font to give the screen a more LCD like appearance. Skeuomorphism may have fallen out of fashion, but it still works in some scenarios; and this is one of them.

First, **download the font from the book's online resources**. The file is called `LCD.ttf`. Place it inside the `MauiCalc` project, in the `Resources/Fonts` folder. We need to wire this up in `MauiProgram` so that we can use it in our code.

To register a font, we can add it to the `ConfigureFonts` extension method on `MauiAppBuilder`. The `IFontCollection` is already passed in to this lambda method, so to add a font we simply call `fonts.AddFont`, and pass it the filename of the font we want to register, and an alias for the font for us to refer to in code.

Open `MauiProgram.cs` and update the font registrations to include our LCD font. Listing 5.6 shows you how to do this, with the added code shown in **bold**.

Listing 5.6 Adding a font registration to MauiProgram

```
namespace MauiCalc;

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");
            fonts.AddFont("LCD.ttf", "LCD");#A
        });
    }

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

Registering fonts

You define which files in your project are fonts by setting their `BuildAction` property to `MauiFont`. This is done using the Properties panel in Visual Studio, or by registering it in the `.csproj` file:

```
<MauiFont Include="[your font file]" />
```

If you look in the MauiCalc.csproj file now, you'll see that there is already an entry matching this, but it has a wildcard for the whole of the Resources/Fonts folder. So, if you want to add a font to your .NET MAUI app, simply copy the font file to this folder, and it will be available for you to register by filename in the `MauiProgram` configuration builder.

If you've come from Xamarin.Forms, you'll note the striking difference in how easy this is, compared to the old way!

Now that we've got a cool LCD font in our calculator, we can add a background color for the screen too. Rather than use one of the pre-defined colors, let's add one the specific color we want to use. To do this, we'll add it to the `ResourceDictionary` in `Resources/Styles.xaml`. We'll look at how this works in more detail in chapter 11, but for now, add the line in **bold** in listing 5.7 to the `Styles.xaml` file.

Listing 5.7 The LCD color in Styles.xaml

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xaml-comp compile="true" ?>
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">

    <Color x:Key="LcdBackgroundColor">#D1E0BA</Color>#A

    <!--code omitted for brevity -->

</ResourceDictionary>
```

Listing 5.8 shows you how to add this color, and the newly imported font, to the LCD Label. The changes are shown in **bold**.

Listing 5.8 The LCD styling in MainPage

```
<?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="MauiCalc.MainPage">
```

```
<Grid ColumnDefinitions="*, *, *, *"
      RowDefinitions="*, *, *, *, *"
      RowSpacing="2"
      ColumnSpacing="2">

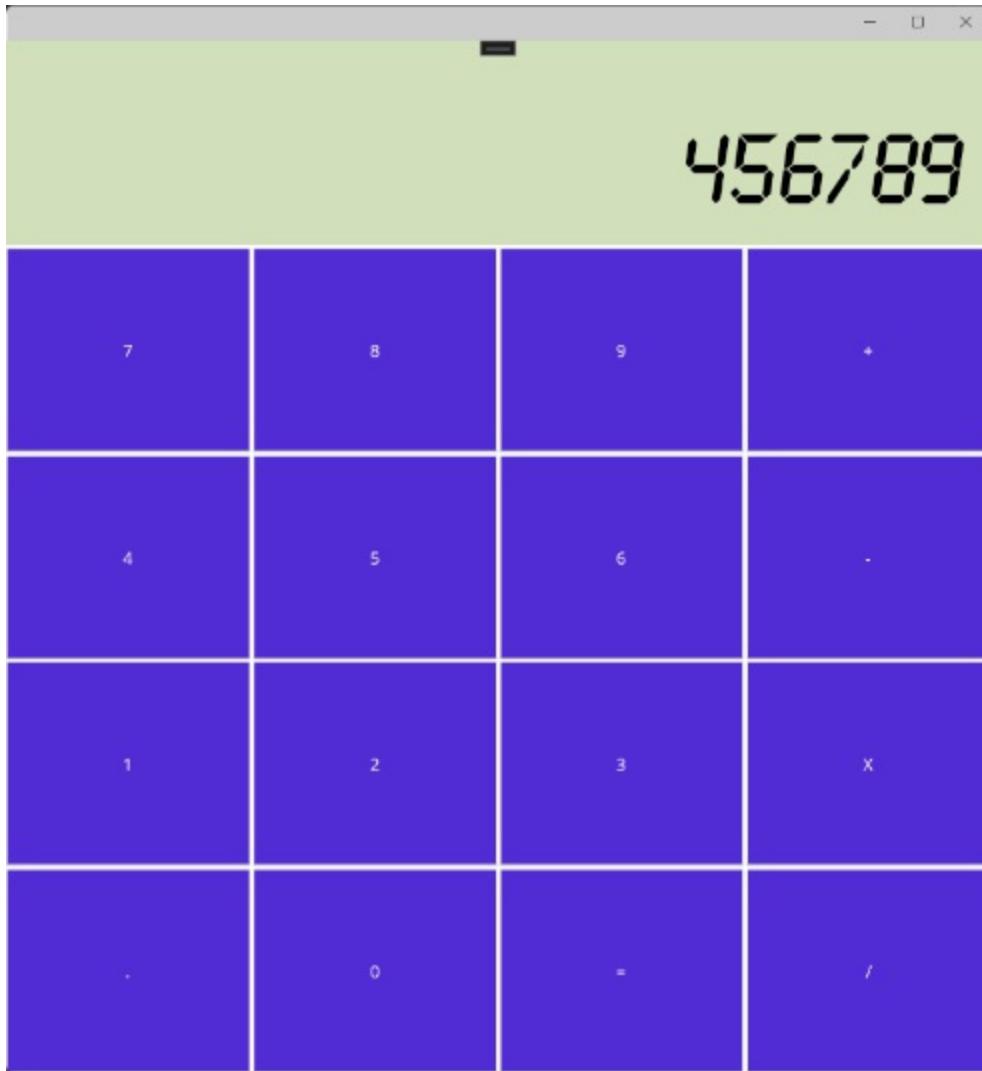
    <Label Grid.Row="0"
          Grid.Column="0"
          Grid.ColumnSpan="4"
          BackgroundColor="{StaticResource LcdBackgroundColor}"
          FontFamily="LCD">#B
          FontSize="72"
          Padding="20"
          TextColor="Black"
          HorizontalTextAlignment="End"
          VerticalTextAlignment="End"
          x:Name="LCD"
          Text="456789"/>#C

    <!--code omitted for brevity -->

  </Grid>
</ContentPage>
```

Run the MauiCalc app again now, and you should get something like figure 5.6.

Figure 5.6 MauiCalc, running on Windows, with the LCD font and background color applied to the Label, to give it a more calculator-like appearance.



We've now completed the UI for the MauiCalc app. Now all that's left to do is make it actually perform some calculations! First, **delete the Text property from the LCD Label control**. Next, update your `MainPage.xaml.cs` file to match Listing 5.9.

Listing 5.9 The full code for `MainPage.xaml.cs`

```
namespace MauiCalc;

public partial class MainPage : ContentPage
{
    public string CurrentInput { get; set; } = String.Empty; #A
    public string RunningTotal { get; set; } = String.Empty; #B
```

```
private string selectedOperator;#C
string[] operators = { "+", "-", "/", "X", "=" };#D
string[] numbers = { "0", "1", "2", "3", "4", "5", "6", "7",
bool resetOnNextInput = false; #F
public MainPage()
{
    InitializeComponent();
}

private void Button_Clicked(object sender, EventArgs e)
{
    var btn = sender as Button;#G
    var thisInput = btn.Text;#H
    if (numbers.Contains(thisInput))#I
    {
        if (resetOnNextInput);#J
        {
            CurrentInput = btn.Text;#K
            resetOnNextInput = false;#L
        }
        else
        {
            CurrentInput += btn.Text;#M
        }
        LCD.Text = CurrentInput;#N
    }
    else if (operators.Contains(thisInput));#O
    {
        var result = PerformCalculation();#P
        if (thisInput == "=");#Q
        {
            CurrentInput = result.ToString();#R
            LCD.Text = CurrentInput;#S
            RunningTotal = String.Empty;#T
            selectedOperator = String.Empty;#U
            resetOnNextInput = true;#V
        }
    }
}
```

```

        }
    else
    {
        RunningTotal = result.ToString();#W
        selectedOperator = thisInput;#X
        CurrentInput = String.Empty;#Y
        LCD.Text = CurrentInput;#Z
    }
}

private double PerformCalculation()
{
    double currentVal;
    double.TryParse(CurrentInput, out currentVal);#AA

    double runningVal;
    double.TryParse(RunningTotal, out runningVal);#BB

    double result;

    switch (selectedOperator)##CC
    {
        case "+":
            result = runningVal + currentVal;
            break;
        case "-":
            result = runningVal - currentVal;
            break;
        case "X":
            result = runningVal * currentVal;
            break;
        case "/":
            result = runningVal / currentVal;
            break;
        default:
            result = currentVal;
            break;
    }

    return result;#DD
}

```

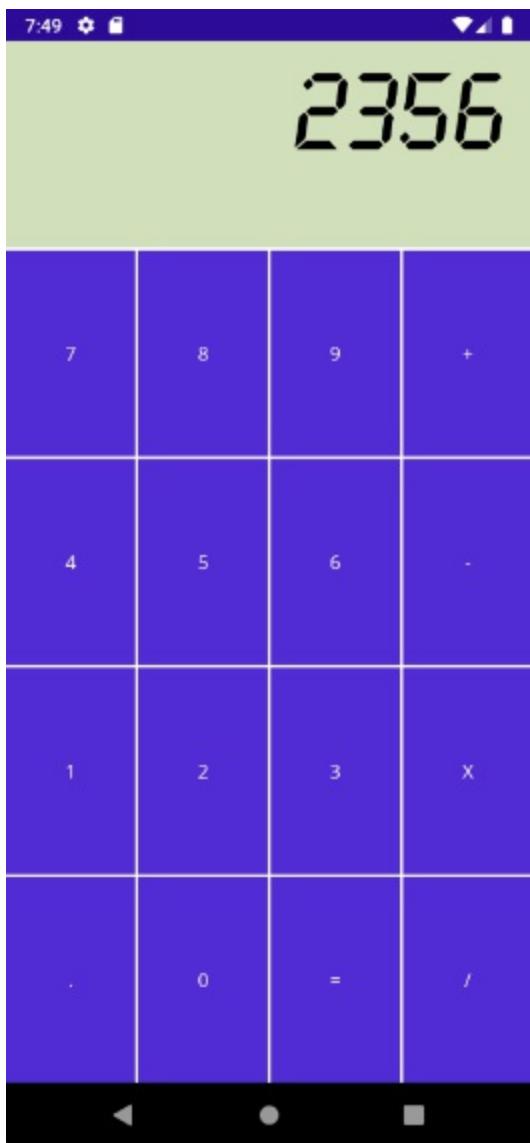
```
}
```

This gives us a fully working version of MauiCalc. Run it now and test it out by performing some calculations.

5.1.3 Row and Column Sizing

If you run the app on a phone (Android or iOS), you will notice that the layout of the Buttons doesn't work as nicely as on desktop (where, in particular, you can resize the window to make them look nicer), or even if on desktop you resize the window to be taller and narrower rather than roughly square.

Figure 5.7 MauiCalc running on Android. Because the rows and columns are proportional, on a portrait orientation device the Buttons look unpleasantly elongated.



Solving this problem is a question of design rather than a technical issue. Ideally, we would design our app to adapt to different orientations and screen proportions. We'll have a look at how to do that in chapter 10, but in the meantime let's look at an alternative layout, still using `Grid`, that might be more portrait friendly.

Figure 5.8 An updated design for MauiCalc, that keeps the Buttons the same size irrespective of the size of the screen



We can achieve this layout using absolute rather than proportional sizing. So far, we've made all our rows and columns an equal share of the total available height or width, but we have two other ways of breaking these down.

The first is to still use proportional sizing but using a ratio rather than an equal share. This allows us to declare that we want any row or column to take up a specific proportion of the available height or width, rather than an equal share. We do this by prepending a number to the asterisk.

For example, let's say we want the first column to take up half the screen, the second to take up a third, and the last column to take up one sixth of the space. For the rows, we want the first row to take up one twelfth of the screen, the middle row to take up three twelfths, and the last row to take up eight twelfths. This can be written using proportional sizing like so:

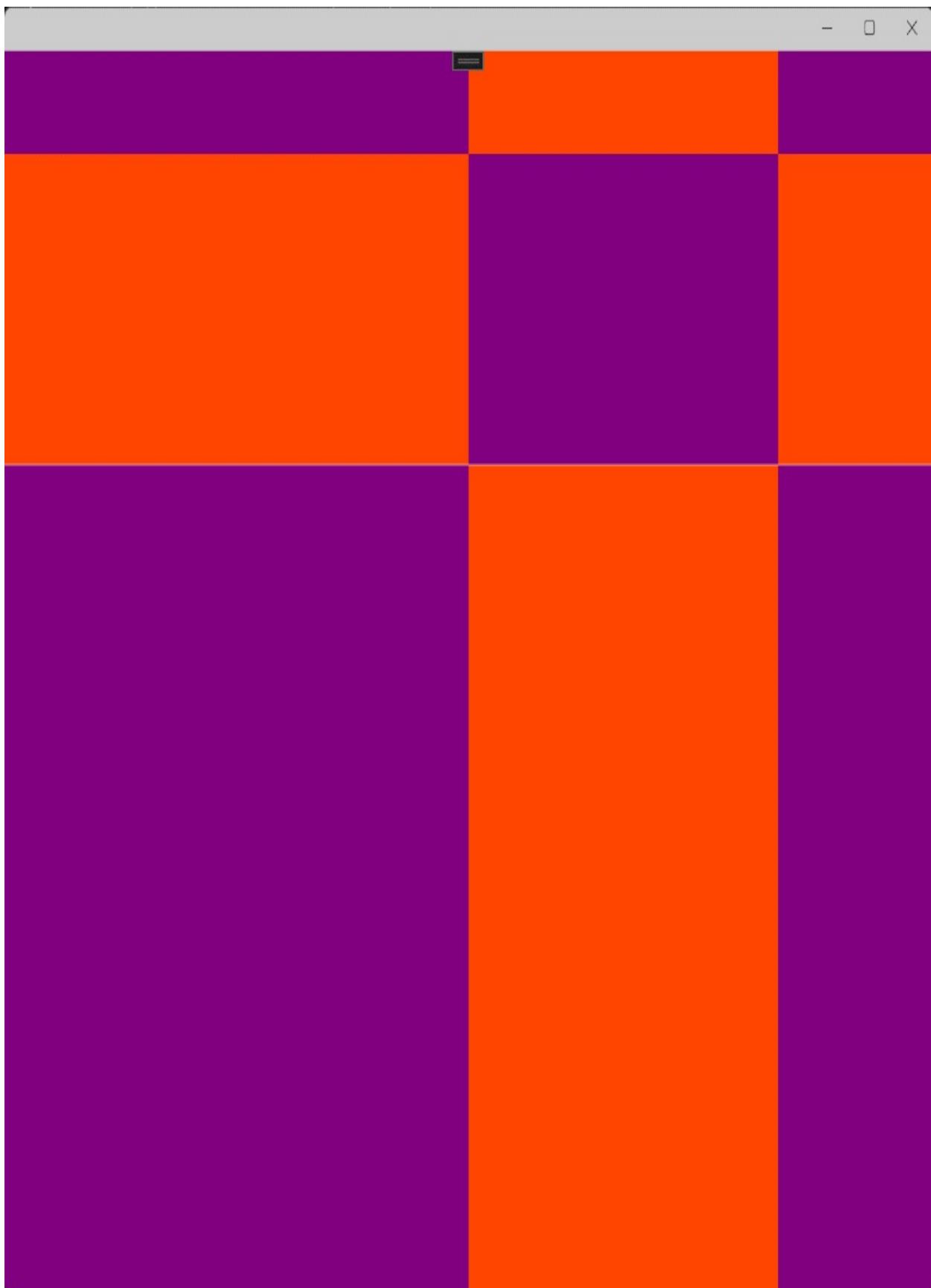
```
ColumDefinitions="3*, 2*, 1*"  
RowDefinitions="1*, 3*, 8*"
```

It's important to remember that these are relative proportions, not DIUs. So the row definitions can also be written as:

```
RowDefinitions="4*, 12*, 32*"
```

And you will get exactly the same result.

Figure 5.9 A Grid with relatively proportioned rows and columns. The first column takes up half the screen, the second column takes up one third, and the last column takes one sixth. The first row takes up one twelfth, the second takes three twelfths, and the last row takes up eight twelfths.



We can use this logic with MauiCalc to get closer to the look that we want. We can make this look better on a phone by specifying that the screen should take up the top half of the available space, and the Buttons can take the remaining space.

We know that we have five rows; one row is the screen and then four rows of Buttons. That means that the screen row needs to be four times as high as any of the Button rows and will therefore take up half the vertical space. We can achieve this by setting the height of the screen row to 4* and leaving all the Button rows as * (it's not necessary to write 1*). Listing 5.10 shows the updated Grid with its new row and column definitions.

Listing 5.10 Updated Grid with proportional screen and rows

```
<Grid ColumnDefinitions="*, *, *, *"  
      RowDefinitions="4*, *, *, *, *"#A  
      RowSpacing="2"  
      ColumnSpacing="2">
```

If you run the updated MauiCalc app now on iOS or Android, you should see something like figure 5.10.

Figure 5.10 The updated MauiCalc app running on Android, with the screen now taking up half the of the available height of the Grid



This is a big improvement; it looks much better than the vertically stretched buttons we had before and opens up some space at the top of the screen where we can display more information, like the running total, memory (if we choose to add a memory button later on), or anything else we can think of.

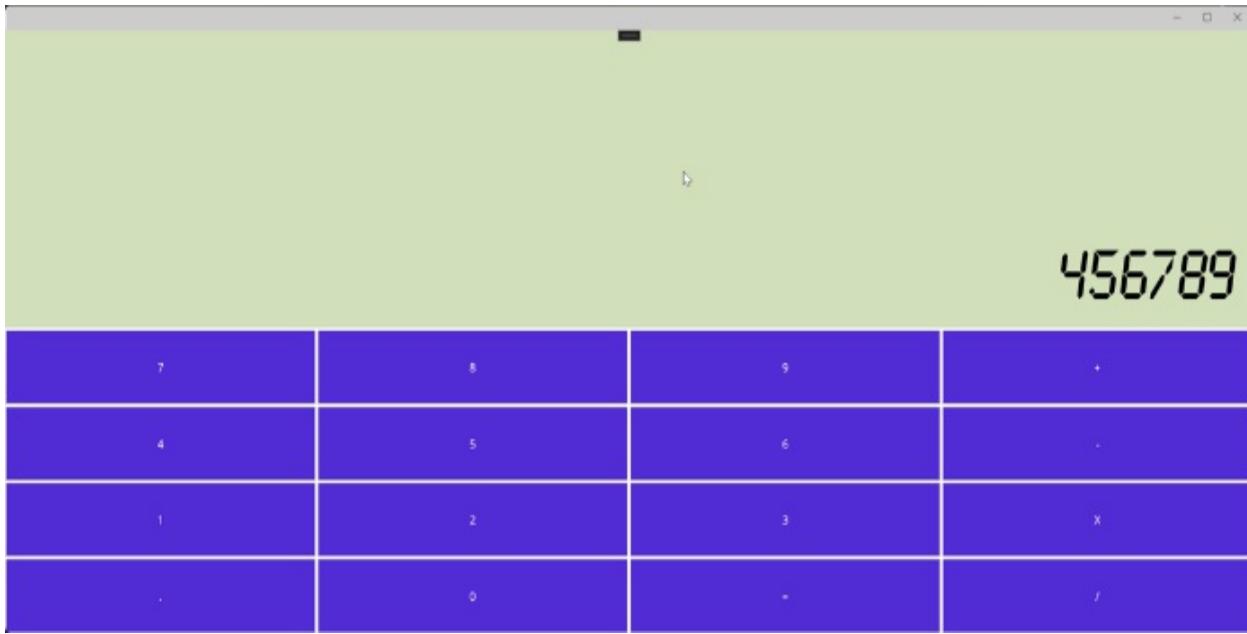
This new layout works well on desktop too, as you can see in figure 5.11.

Figure 5.11 MauiCalc running on Windows, with the rows proportioned so that the screen takes up half the available height



This looks ok, although we can already see that now we have the reverse problem – the buttons are now stretched horizontally. And let's not forget that a window on a desktop OS can be resized, so this could end up looking like figure 5.12.

Figure 5.12 MauiCalc running on Windows, with the rows proportioned so that the screen takes up half the available height, but with the window stretched horizontally.



This still doesn't give us the sketched out design from figure 5.8. To do that we need to combine the proportional sizing we've been using so far with absolute sizing.

Absolute sizing is easy; instead of specifying a proportion, you specify a value in DIUs. With this approach, we can specify the height and width of each row and column, which means we can make every `Button` exactly square.

Let's update the row and column definitions. We'll make each column 100 DIUs wide, and each `Button` row 100 DIUs tall. We can then set the screen row back to `*` to let it take up any remaining available height. Listing 5.11 shows the updated definitions. The changed lines are shown in **bold**.

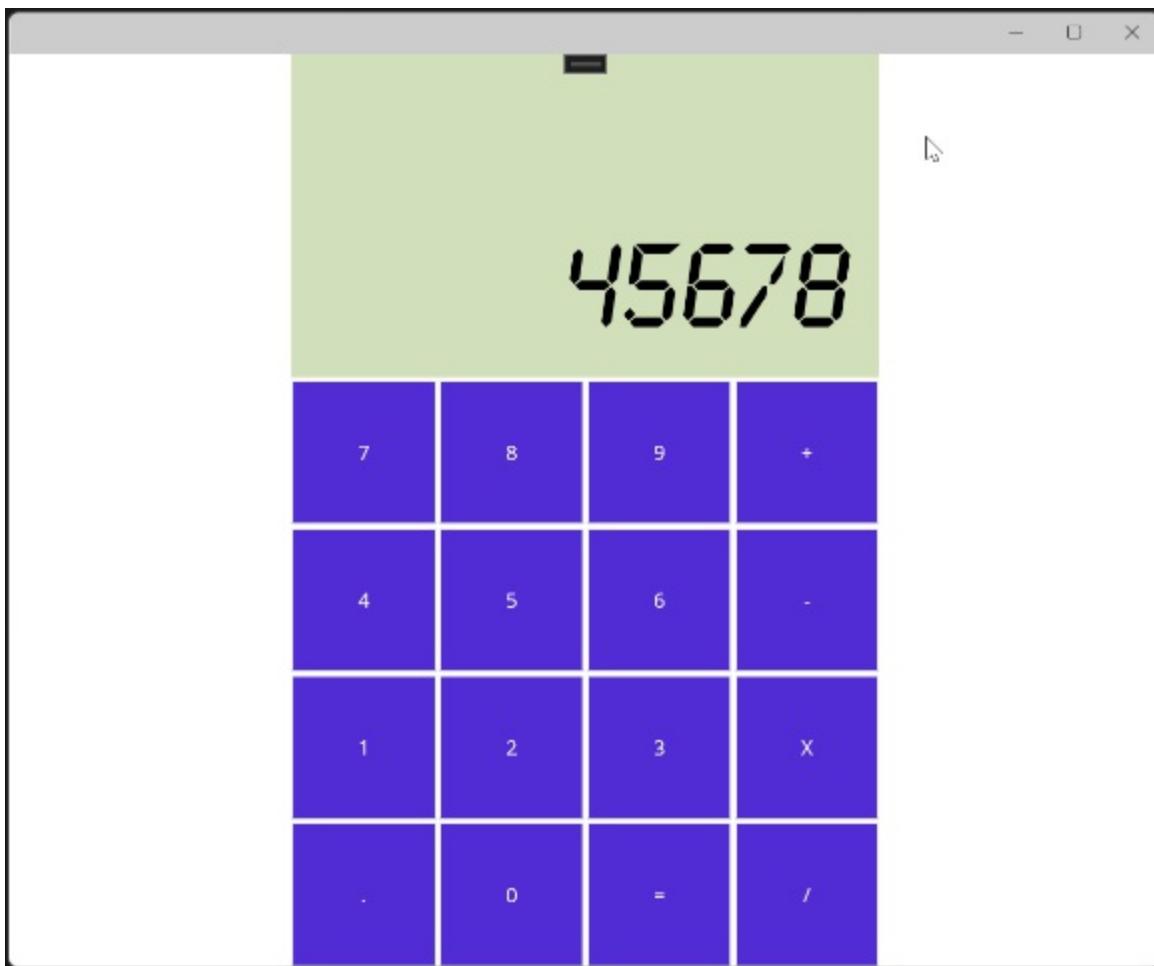
Listing 5.11 MauiCalc with fixed height and width buttons

```
<Grid ColumnDefinitions="100,100,100,100">#A  
    RowDefinitions="*,100,100,100,100"#B  
    HorizontalOptions="Center"#C  
    RowSpacing="2"  
    ColumnSpacing="2">
```

If you run this now on a phone, it will look roughly the same as it did before. If you run it on desktop, you'll see that we got the design we intended, but it

still doesn't look great.

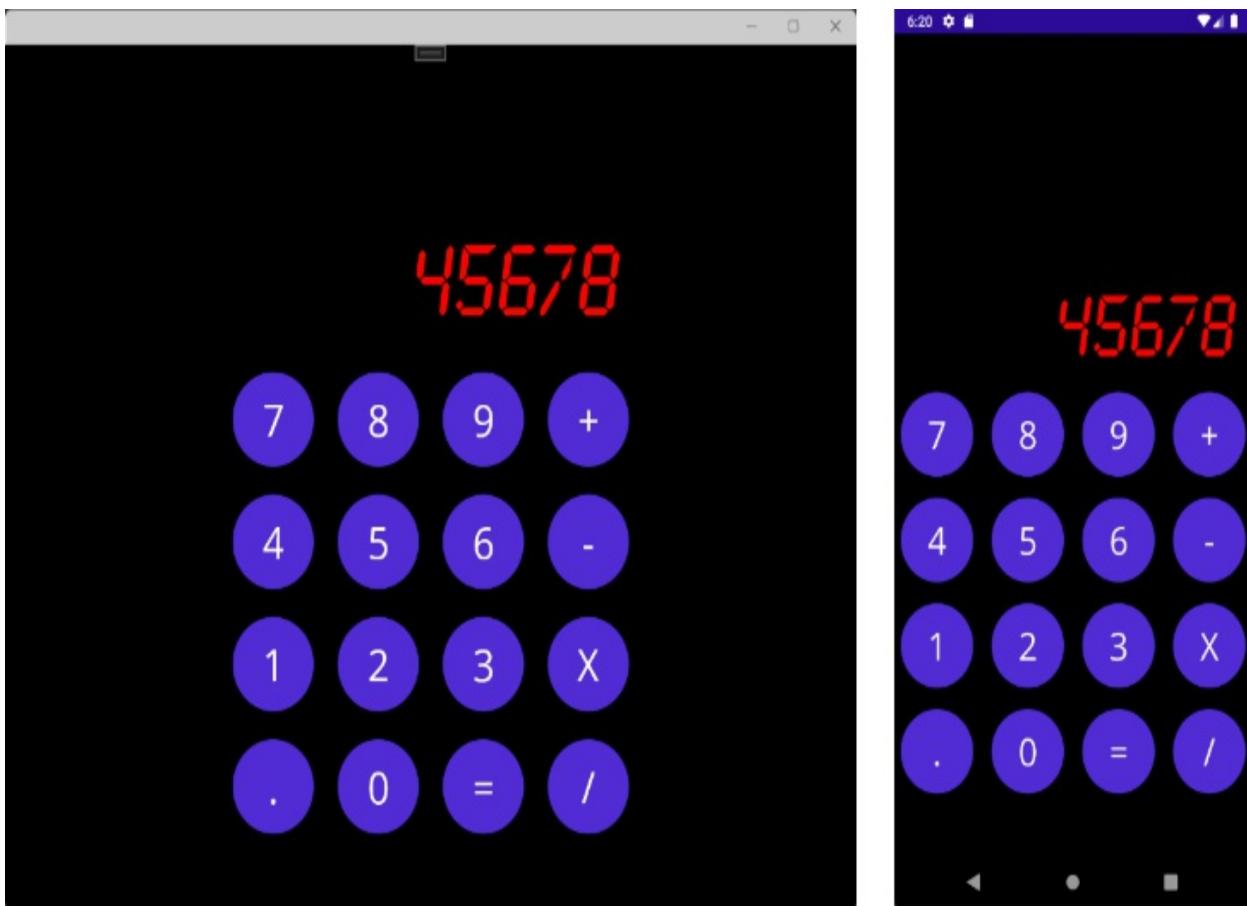
Figure 5.13 MauiCalc running on Windows with fixed height and width Buttons. It now matches the design we intended, providing a consistent look across desktop and mobile. But there is room for improvement.



Nevertheless, this matches our design goal, and improving the look is a design challenge rather than a technical one. The layout is now correct, and we have used Grid to achieve what we wanted to achieve.

You can make some other design choices to improve this, for example, changing the background color and LCD font color, and adding a corner radius to the Buttons can give you something like this:

Figure 5.14 FancyCalc – the MauiCalc app with some small changes to the UI, but the layout is the same



You can find the source code for this design in the FancyCalc folder in the book's online code resources, but it's not listed here as it's outside the scope of the `Grid` discussion. The key point to remember is that the changes here are changes to the controls; the layout remains unchanged.

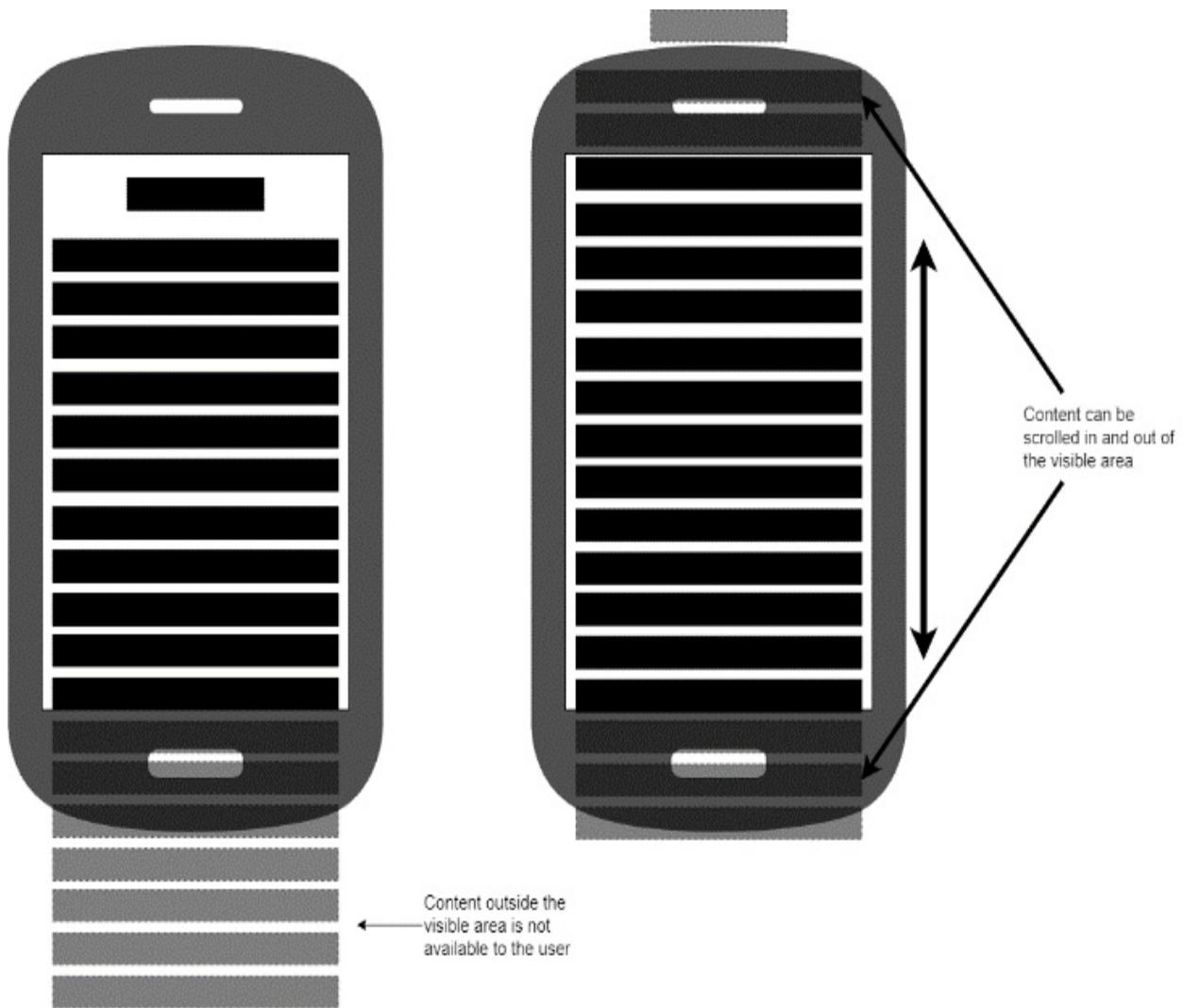
We'll look more at accommodating different platforms and screen sizes in chapter 10.

5.2 ScrollView

`ScrollView` is a control that allows its child content to be scrolled. It's useful in cases where you have content of indeterminate length and need to provide a way to present more content to the user than can fit on the screen. Figure 5.15 shows an example of this.

Figure 5.15 Large amount of text shown on the left in a `VerticalStackLayout`. The content is too long to fit on the screen, so the user is only able to read the first few lines. On the right, the same

text is shown in a ScrollView, so the user can swipe up and down to see the full body of the text.



In figure 5.15, a ScrollView is used to enable users to read paragraphs of text which don't fit on the screen. Without it, as in the example on the left, with the text rendered in a VerticalStackLayout, only the content that fits on screen is available to users.

Note that by default the orientation of the ScrollView (i.e., the direction of scrolling) is vertical. You can also specify the direction as horizontal if you wish.

Vertical and horizontal aren't the only options. You can also set the orientation to both or neither. Both, as the name suggests, allows the user to scroll both vertically and horizontally. This could be useful for showing an

image at full scale and allowing the user to pan around it. It can also be a useful accessibility feature; some users may prefer large type, and being able to scroll the text both vertically and horizontally can be preferable for some users to scrolling vertically through many lines of just one or two words.

Take care with nested scrolling views

Be careful when nesting scrolling views inside each other as it can make the UI difficult to use, or impossible in some cases.

An obvious example would be to not place one `ScrollView` inside another. Some more subtle examples could include a `CollectionView` inside a `ScrollView`, or a `ScrollView` in a bottom sheet.

As the user is swiping their finger up or down the screen, the results can be unexpected as your app won't necessarily scroll the view that your user is expecting to scroll; so perhaps a more precise warning is to not use overlapping control gestures.

If you find yourself using nested scrolling views, rethink your design so that your scrolling views only appear on fixed pages.

The exception is scrolling views where their direction of scroll is perpendicular. If for example you have a page that scrolls vertically, you can include a horizontally scrolling `CollectionView` in the page, as there is no ambiguity between directions of scroll for the gestures your user provides.

A popular trend is to use `ScrollView` as an enclosing container for pages. Because devices come in many different shapes and sizes now, it has become impossible for designers to create a UI that looks the same on all kinds of screen aspect ratios. The depth of this problem is illustrated by the design trend away from “pixel perfect” designs toward design systems. There’s a neat summary of this issue you can read here:

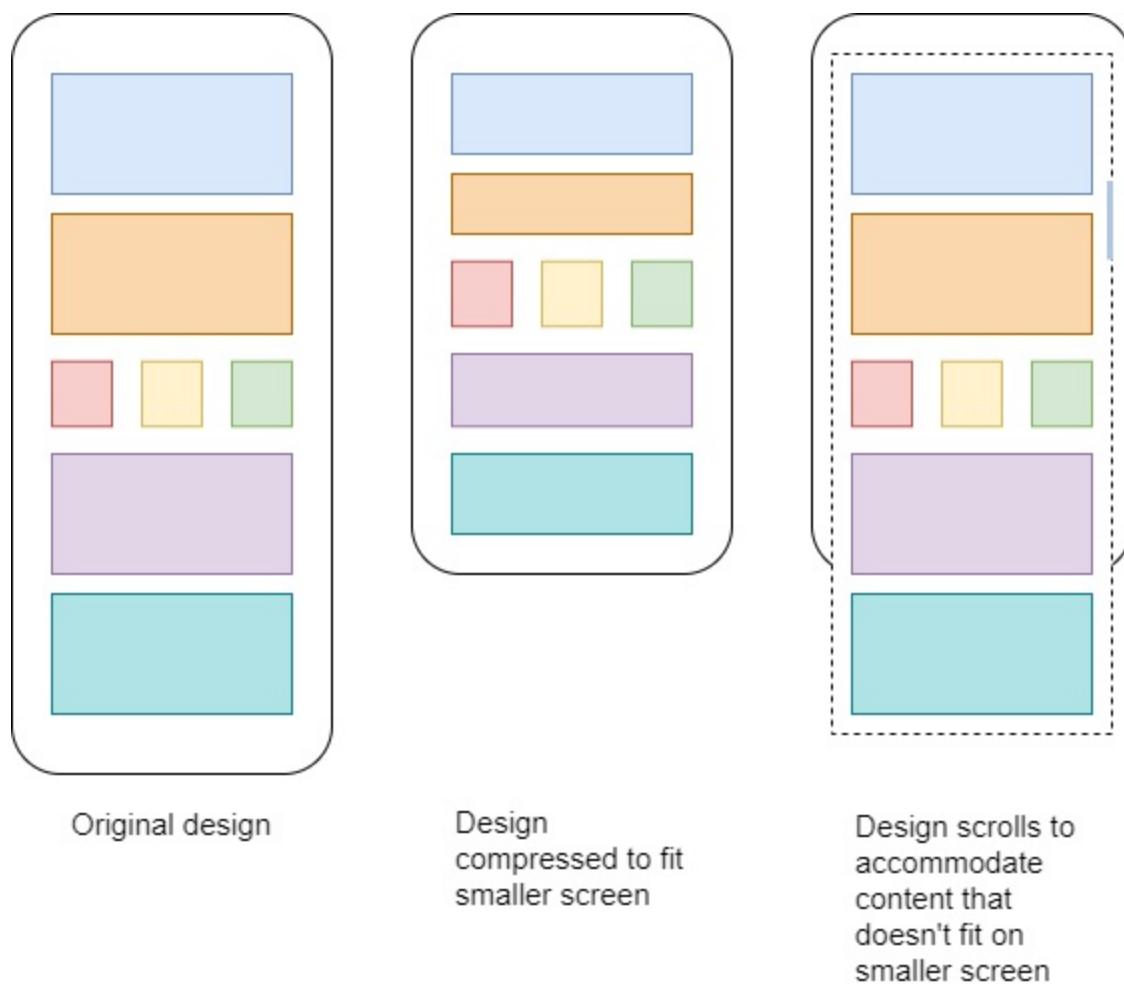
<https://www.kelliekowalski.com/articles/the-myth-of-pixel-perfection>, and a quick web search for the term “pixel perfect myth” will turn up plenty of results. You’ll get an appreciation for the problem.

Note

This “pixel perfect myth” is a good reason why the default approach with .NET MAUI (consistent but not identical, as discussed in chapter 1) can often lead to much better UI and UX. That’s not to say you can’t achieve a “pixel perfect” UI with .NET MAUI (insofar as the term is commonly accepted nowadays); but it’s well worth considering whether delivering a consistent experience is more important.

Consider the example in figure 5.16. This design has a bunch of cards in a vertical stack, with a horizontally scrolling collection in the middle.

Figure 5.16 a mobile UI design created for a long screen (left). When rendering this UI on a screen with a different aspect ratio that makes the device less ‘tall’, you have two options. The first is to squish the content so that it all fits on screen (middle). The second is to enclose the content in a ScrollView (right), so that the content appears the same size as the original design, and the user can scroll to see content that doesn’t fit on screen.



The original design, shown on the left, is based around a tall and narrow screen. To adapt it for a squatter, shorter screen, we could either vertically compress the visual elements, resulting in something like the middle option, or we could keep them all the same size and shape, but make the whole view scroll, as depicted on the right. We've seen this approach already; it's in the template that we used to build all our apps so far. You can see it in `MainPage.xaml` in the Aloha, World! app.

A fourth option would be to design a responsive UI that has space for visual elements to be repositioned without compromising the elements themselves. This approach works up to a point but has its limits. We'll have a look at this approach in chapter 10.

You can choose the approach that works best for you, although the scrolling option is gaining popularity, and the `ScrollView` in .NET MAUI lets you implement this approach with minimal effort.

5.3 HorizontalStackLayout and VerticalStackLayout

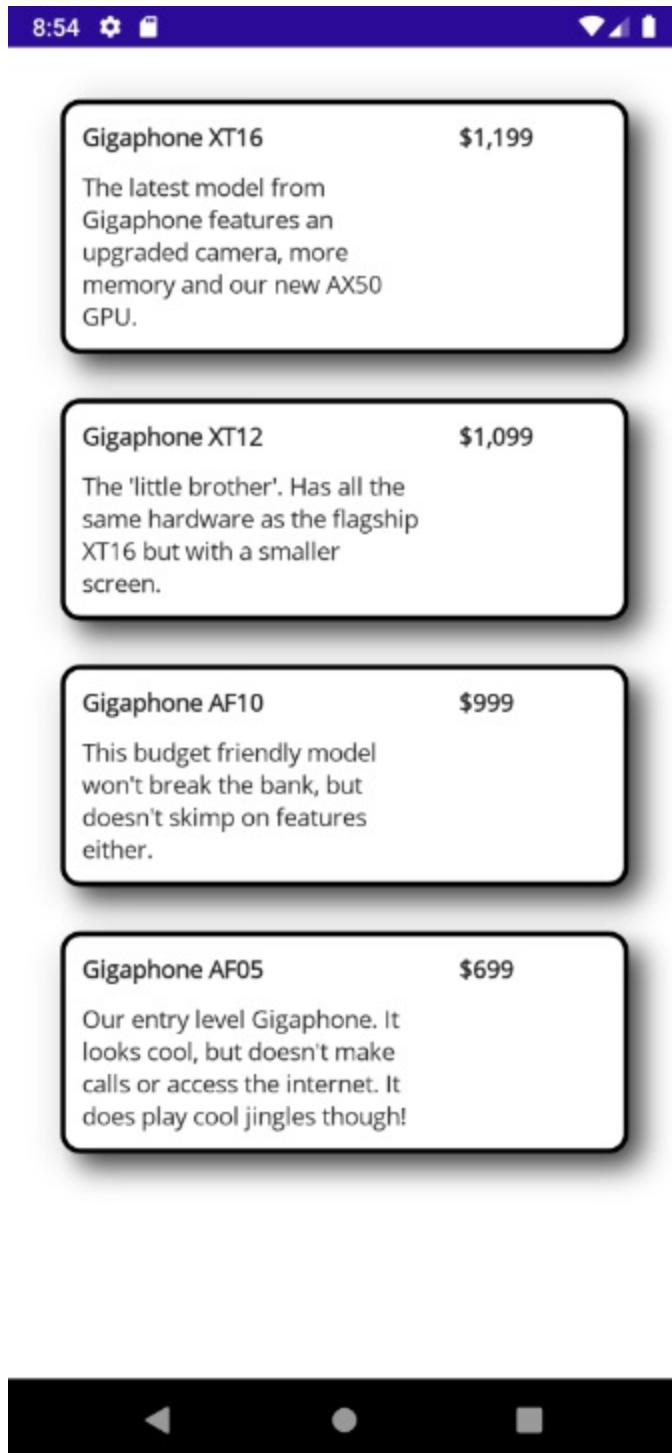
The two stack layouts in .NET MAUI are the simplest layouts to use, and probably the easiest for building quick prototypes or trivial apps.

`VerticalStackLayout` and `HorizontalStackLayout` work in the same way, except for their orientation. They have a collection of type `View` called `Children`, and `Views` are added to the stack and rendered on-screen in the order in which they are added. This means you can add any control or layout to a `VerticalStackLayout` or `HorizontalStackLayout`.

We've seen `VerticalStackLayout` in use already; it's also in the template we've been using and is nested inside the `ScrollView`. You can see it in `MainPage.xaml` in the Aloha, World! app.

You can combine `VerticalStackLayout` and `HorizontalStackLayout` to create almost any UI. As an example, let's reimagine CellBoutique using cards for the products rather than a `CarouselView`, as in figure 5.17.

Figure 5.17 A collection of cards displaying product information. The cards are displayed in a vertical stack, and each card uses a vertical and horizontal stack to layout its content.

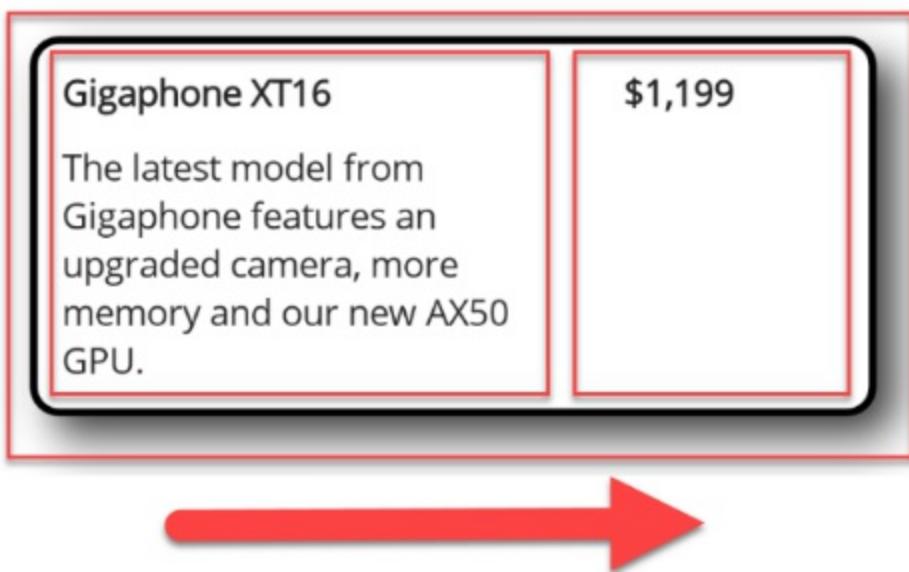


In this example, the `CarouselView` has been replaced with a `CollectionView` (we could also just use a `VerticalStackLayout` inside a `ScrollView` and

hard-code the products), and the card template is built entirely using `VerticalStackLayout` and `HorizontalStackLayout`. Let's break it down.

Each card will itself be a `HorizontalStackLayout`, which will allow it to arrange its children horizontally, from left to right. The children will be a `VerticalStackLayout` to display the product title and description, and a `Label` to display the price.

Figure 5.18 The cards themselves are a `HorizontalStackLayout` that arrange their child views horizontally on screen, from left to right.



The first child view of the `HorizontalStackLayout` that makes up the card is a `VerticalStackLayout` that's used to arrange the product title and description, one above the other.

Figure 5.19 A `VerticalStackLayout` is used to arrange the product title and description vertically, from top to bottom



I've mentioned the importance of layout already; in fact, we can remove all the borders from these cards and rely on layout alone and still have a decent UI. We achieve this using spacing. `HorizontalStackLayout` and `VerticalStackLayout` provide a `Spacing` property, which is used to provide a gap between child views in a stack.

Spacing is one of (if not the) most important aspects of layout. It helps to define hierarchy in your UI and relationships between different elements. If we take our current card UI example, we can remove the borders and shadows and rely only on spacing, as in figure 5.20.

Figure 5.20 This UI uses a combination of `VerticalStackLayout` and `HorizontalStackLayout`. It doesn't have any borders or shadows, but instead uses spacing to group elements.



\$1,199 Gigaphone XT16

The latest model from
Gigaphone features an
upgraded camera, more
memory and our new AX50
GPU.

\$1,099 Gigaphone XT12

The 'little brother'. Has all the
same hardware as the flagship
XT16 but with a smaller
screen.

\$999 Gigaphone AF10

This budget friendly model
won't break the bank, but
doesn't skimp on features
either.

\$699 Gigaphone AF05

Our entry level Gigaphone. It
looks cool, but doesn't make
calls or access the internet. It
does play cool jingles though!



In this example, the borders and shading have been removed. The price has been moved to the left as, without the border, it appears to float without reference. Moving it to the left helps to better define the relationship between the price and the other elements. The space between cards has been increased, and the difference in spacing between each group of elements, and the elements within the group, leaves no ambiguity about which price, title and description belong together.

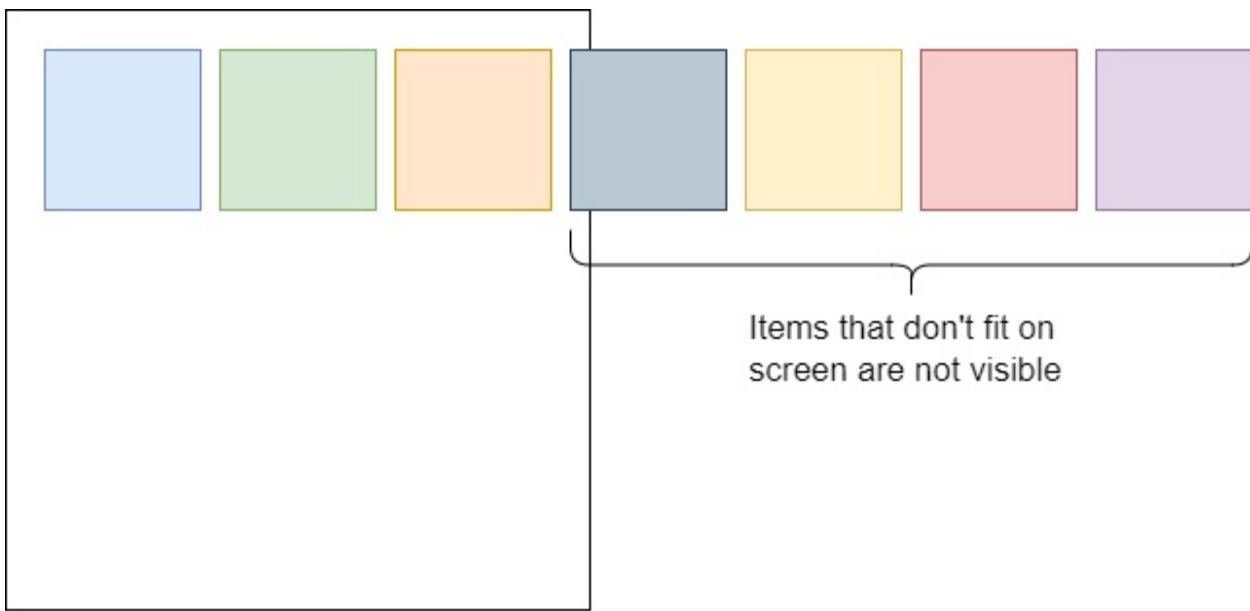
Have a go at refactoring the CellBoutique app to match both of these designs, using the `Borders` and `Shadows` that we learned about in chapter 4, and try the layout without borders too. Remember that the cards are only using `HorizontalStackLayout` and `VerticalStackLayout`. You can see the code for my version in the book's online resources.

5.4 FlexLayout

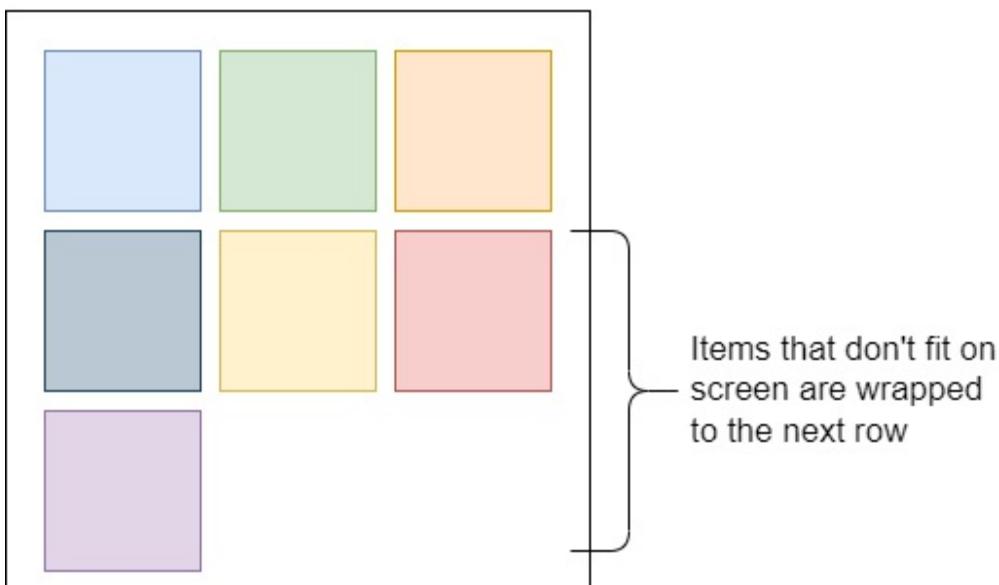
If you're familiar with web technologies and CSS, you've no doubt heard of the flexible box layout, often just called flexbox. In .NET MAUI, we have a similar layout called `FlexLayout`.

`FlexLayout` has a lot of similarities with `HorizontalStackLayout` and `VerticalStackLayout`, but with some important distinctions. The most important difference is that with the two stack layouts, anything that doesn't fit on screen will simply not be rendered and is lost to the UI, but `FlexLayout` will wrap items to the next row or column, depending on the direction specified for the `FlexLayout`.

Figure 5.21 A `HorizontalStackLayout` is used to arrange child items in the top example. Child items that don't fit on screen are simply not visible. The bottom example arranges the same child items in a `FlexLayout` with its `Direction` set to `Row`. Items that don't fit on screen are wrapped to the next row.



HorizontalStackLayout



FlexLayout

`FlexLayout` is almost a direct translation of the CSS flexbox and has similar properties to allow you to arrange child items in either rows or columns and specify the spacing and alignment. We'll look at two different ways of using `FlexLayout` in the next chapter.

5.5 Summary

- Grid is a powerful layout, and you can use it to create almost any UI. Grid is used to arrange child views in rows and columns.
- You can use a ScrollView to make more content than fits on screen available to users. You can use ScrollView for a scrolling section of a page, or even wrap the whole page in a ScrollView to accommodate different screen sizes.
- HorizontalStackLayout and VerticalStackLayout are simple layouts for arrange child views one after the other, vertically or horizontally. Most simple layouts can be accomplished using a combination of these.
- You can combine layouts in .NET MAUI, and in your real-world apps you will likely do so. Combining these is the best way to build the UIs your designers give you in .NET MAUI apps.

6 Advanced layout concepts

In this chapter:

- Using Grids for advanced layouts
- BindableLayout
- AbsoluteLayout
- Combining layouts to build rich UIs

So far, we've looked at `Grid`, `HorizontalStackLayout` and `VerticalStackLayout`, and `FlexLayout`, and you'll likely find that nearly any UI you're trying to achieve can be accomplished using these. But there are more layout options available in .NET MAUI.

Before we get into new layouts, we'll revisit the `Grid` and see how we can apply it to more complex layouts. In chapter 4 we saw how to use it for a UI that naturally lends itself to a grid pattern, but it's also often the best layout for arranging things that don't at first seem to obviously be a grid. We'll work through an example of a UI using `Grid` in a real-world app to get a better understanding of how flexible and powerful it is.

Sometimes, though, you need just a little bit more, and in .NET MAUI there's still plenty of room to grow beyond the basics we've explored so far. In this chapter, we'll look at how you can use `BindableLayout` to convert any layout into a collection view, and we'll see how `FlexLayout` is an ideal candidate for this treatment.

After that, we'll move onto `AbsoluteLayout`. `AbsoluteLayout` can be a powerful tool when you need to hit an exact UI target, and we'll see how you can use it for more exact positioning of elements on a screen.

6.1 Thinking in Grids

We've seen how we can use a `Grid` to lay things out that naturally lend themselves to a grid pattern, but as I mentioned in chapter 4, `Grid` is a

powerful layout, capable of much more than just displaying squares and rectangles.

When it comes to the overall page or screen layout, Grid is the best option for most scenarios. You can achieve nearly any UI using Grid. Let's look at a real-world example to see Grid in action.

SSW Rewards is an app developed by my company to drive engagement with the developer community. Users can accumulate points and exchange them for prizes, and it also features a profiles section that provides a picture and bio of SSW staff. It is open source and people are welcome to adapt it for their own uses.

Figure 6.1 shows the design for the profiles page in the app.

Figure 6.1 The Profiles page of the SSW Rewards app. There are multiple components to this page, and while it may not immediately look like a grid, the Grid layout is used to implement this design.

...

Adam Cogan

SSW Chief Architect, Microsoft
Regional Director

Scrum

Azure

Azure DevOps

YouTube

SSW CHINA!
How many users do you have in China?

GitHub LinkedIn Twitter

Adam Cogan is the Chief Architect at SSW, a Microsoft Certified Gold Partner specializing in Azure, Azure DevOps, .NET, SharePoint and BI solutions. At SSW, Adam has been developing custom solutions for businesses across a range of industries including Government, engineering, banking, insurance, and manufacturing since 1990.

Home

Profile

Scrum

Achievements

Help

< >

You might not immediately look at this and see a `Grid`, but once you've been developing .NET MAUI apps for a while, you'll start to see rows and columns in designs like this, almost like Neo seeing the world as green code rain in *The Matrix*. Let's have a look at how we can start breaking this down.

Figure 6.2 The first step in building out this design is to identify the topmost grid. We can break this design into three rows; a header section, a body section, and a bio section. And we can see that the top row has three columns.

< Adam Cogan SSW Chief Architect, Microsoft Regional Director >

Scrum

Azure

Azure DevOps

YouTube

SSW CHINAFY
How many users do you have in China?

300

[Twitter](#) [GitHub](#) [LinkedIn](#)

Adam Cogan is the Chief Architect at SSW, a Microsoft Certified Gold Partner specializing in Azure, Azure DevOps, .NET, SharePoint and BI solutions. At SSW, Adam has been developing custom solutions for businesses across a range of industries including Government, engineering, banking, insurance, and manufacturing since 1990.

The top level of this design is the page, so let's start there. The page is comprised of three main sections: the header, which contains the name and

title and some navigation buttons, the main details section, which contains a picture, a skills summary, a QR indicator (it shows whether the user has scanned this developer and accumulated the points), and some social interaction buttons.

The first step is easy then – we know we need three rows. You can approximate their relative sizes from this picture; although because this design was shared with me by the UX team in a design collaboration tool, I have the luxury of knowing the precise relative proportions of these rows. They translate to 2*, 9* and 3*.

Now we need to determine the columns. Only the top row uses the columns, but that's not an issue, seeing as we can set the `ColumnSpan` of the remaining two rows to 3, so that they will take up all three columns. The navigation controls can then be placed in columns 0 and 2 of row 0.

The column definitions based on the design are *, 5*, *.

Next, we need to break down the developer details section of this page, and we can also use a `Grid` for this.

Figure 6.3 The grid layout breakdown of the developer details section. We can see that it will have two equally sized columns, with the picture taking up all three rows of the first column. It will also have a ColumnSpan of 2 so that the picture can overlap other items. The remaining items will be in the second column, with the skills summary in the first row, the QR icon in the second, and the social interaction buttons in the third.

Adam Cogan

SSW Chief Architect, Microsoft
Regional Director

The image is a composite of two parts. On the left is a photograph of Adam Cogan, a man with long blonde hair, wearing a red t-shirt with a graphic that includes the text "SSW CHINAFY" and "How many users do you have in China?". He is smiling and giving a thumbs-up with both hands. On the right is a vertical column of four colored boxes with white text: a red box for "Scrum", a dark red box for "Azure", a light red box for "Azure DevOps", and a dark grey box for "YouTube". Below these boxes is a QR code icon followed by the number "300" and a yellow star icon. At the bottom are three social media icons: Twitter, GitHub, and LinkedIn.

Adam Cogan is the Chief Architect at SSW, a Microsoft Certified Gold Partner specializing in Azure, Azure DevOps, .NET, SharePoint and BI solutions. At SSW, Adam has been developing custom solutions for businesses across a range of industries including Government, engineering, banking, insurance, and manufacturing since 1990.

This section will have two columns of equal width. The picture will be in the left column, and everything else will be in the right column. In the right column, the top row will hold the skills summary, the middle row will hold the QR icon, and the bottom row will hold the social interaction buttons.

Determining the relative column widths is straightforward as they both take up an equal share of the width, so they are both `*`. For the rows, from the design I can determine their relative heights of `5*`, `2*` and `*`.

Listing 6.1 shows these Grids, with their row and column definitions.

Listing 6.1 The Grid layout for the SSW Rewards People page

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    BackgroundColor="{StaticResource PeopleBackground}"
    xmlns:controls="clr-namespace:SSW.Rewards.Controls"
    xmlns:converters="clr-namespace:SSW.Rewards.Converters"
    x:Class="SSW.Rewards.Pages.PeoplePage">
    <ContentPage.Content>
        <Grid RowDefinitions="2*, 9*, 3*"
            ColumnDefinitions="*, 5*, *">
            <Grid Grid.Row="1"
                Grid.Column="0"
                Grid.ColumnSpan="3"
                ColumnDefinitions="*, *"
                RowDefinitions="5*, 2*, *">
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

Listing 6.1 shows only the `Grid` definitions; it's a complex page and including all of the views, converters, helpers and bindings would be out of scope of this section. However, as I mentioned above, it's open source so if you want to look at the full source code, you can see it on GitHub at <https://github.com/SSWConsulting/SSW.Rewards>. We'll also revisit SSW Rewards in appendix B when we look at using the .NET Upgrade Assistant to port a Xamarin.Forms app to .NET MAUI.

We'll come back to `Grids` with a hands-on example at the end of this chapter.

6.2 BindableLayout

`BindableLayout` isn't a layout itself. It's a static class that can be attached to

any layout that will allow it to generate its own content. Essentially, it turns any layout into a collection view.

Using `BindableLayout`, we can specify an `ItemsSource` property for a layout and then use a `DataTemplate` to render each item in the collection, just like with `CollectionView`, as if they were hard-coded child views of the layout we attach `BindableLayout` to. This can work well with `HorizontalStackLayout` or `VerticalStackLayout`, or particularly well with `FlexLayout` as we'll see in an example shortly. It doesn't make much sense to use `BindableLayout` with `Grid`, as views in a `Grid` need their `Row` and `Column` properties to arrange themselves properly.

Using `BindableLayout` with one of the stack layouts or with `FlexLayout` can make it similar to `CollectionView`, but `CollectionView` offers functionality you don't get with `BindableLayout`, for example the ability to select one or more items, backed by bindable properties. `BindableLayout` is a good option when just need a data source for rendering content; `CollectionView` is a better option when you need your user to interact with the collection, rather than just the items in that collection.

Let's see how we can use `BindableLayout` with `FlexLayout` to arrange a collection of items on screen. We're going to build an app to give us movie recommendations, based on what's currently trending and our chosen genres.

The UI will be fairly simple. The user will see a list of trending movies and will have an option to filter the list by genre. Figures 6.4 to 6.6 show the mock-ups for the `MauiMovies` app.

Figure 6.4 The main UI layout for the `MauiMovies` app. A list of trending movies is shown, with a poster, title, and rating. At the top of the screen is a list of genres the user has chosen to filter by. The user can tap on this list to change their selection.

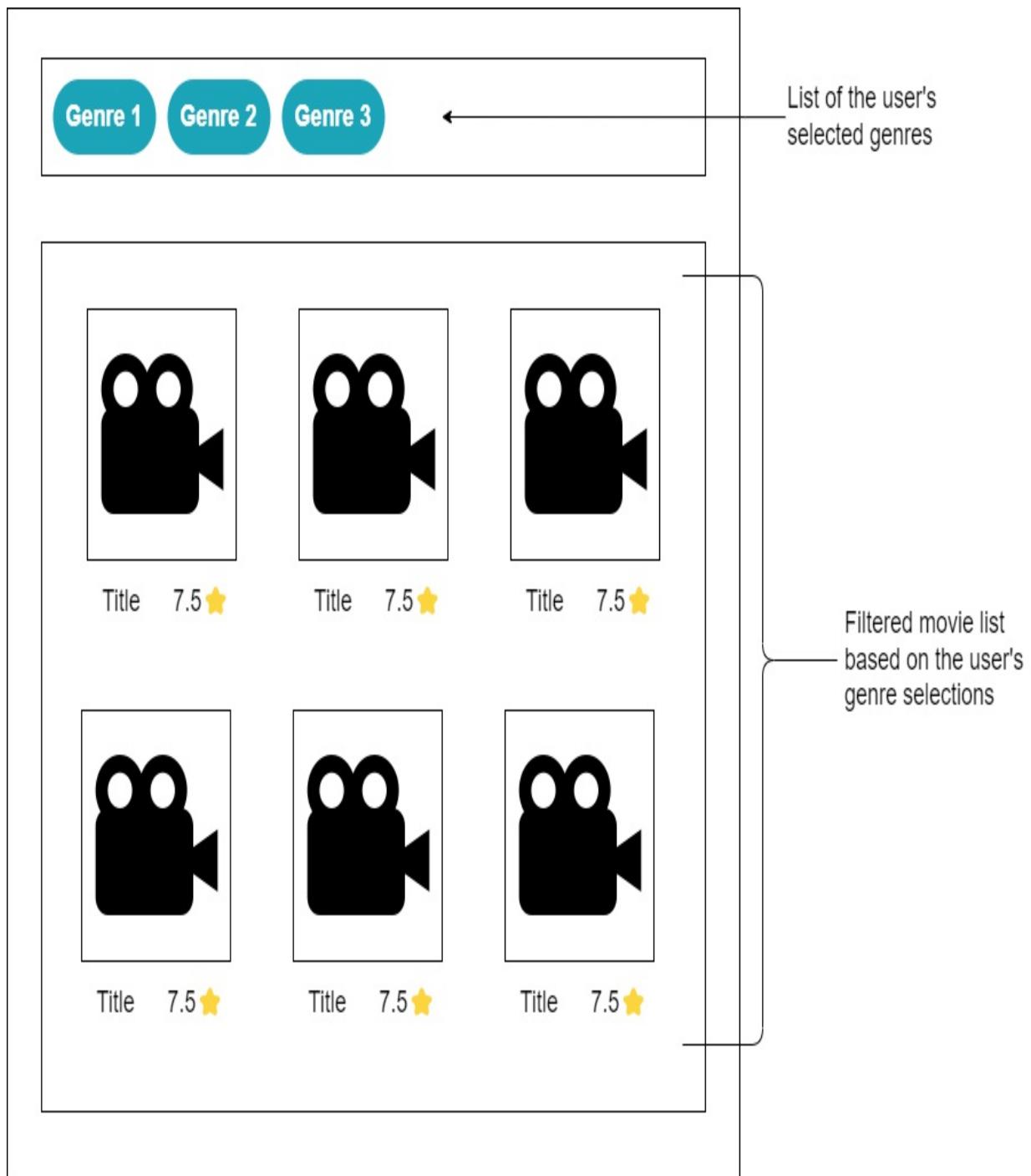


Figure 6.5 The user can tap the chips showing the selected genres to see the list and change their selection.

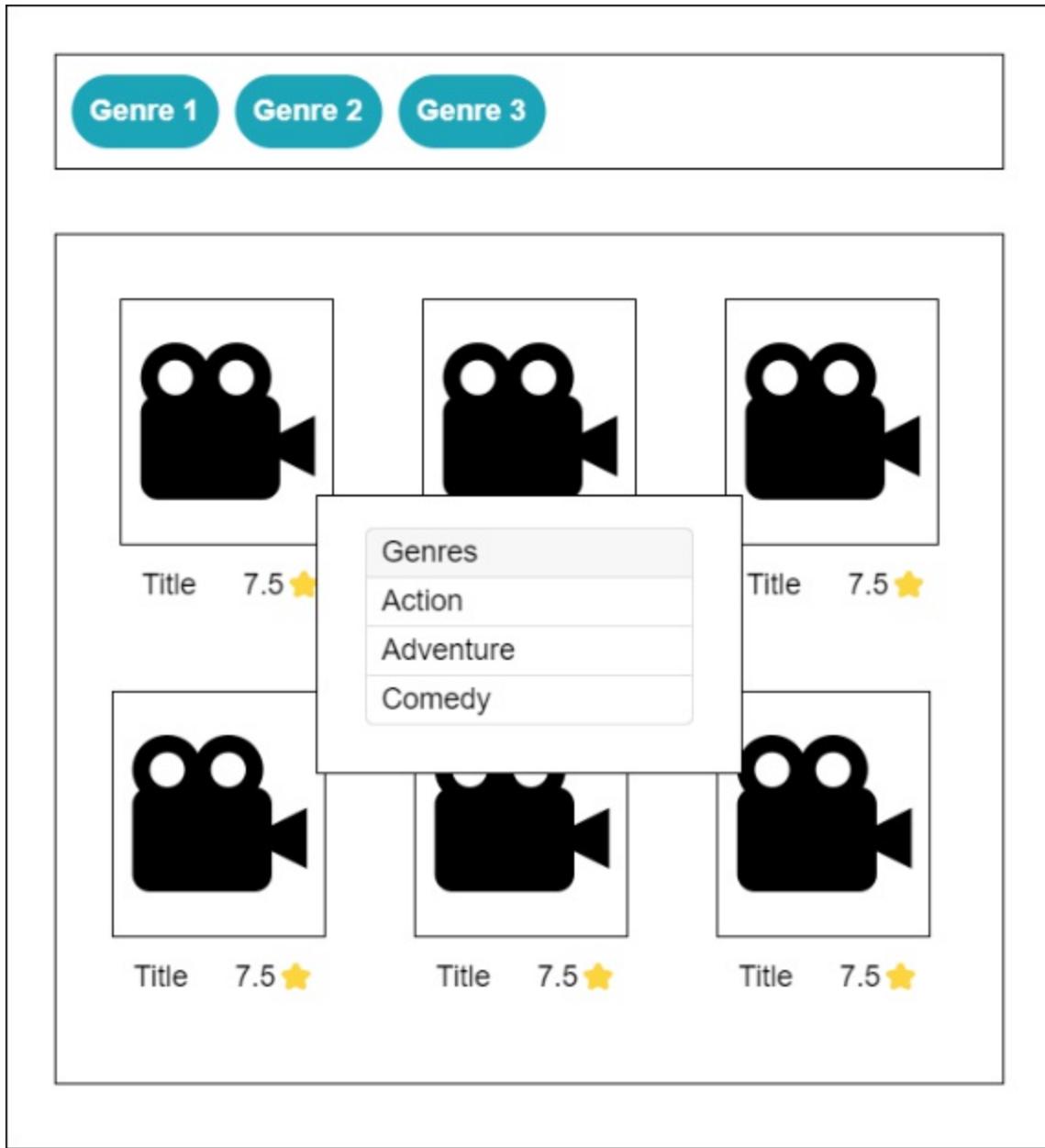
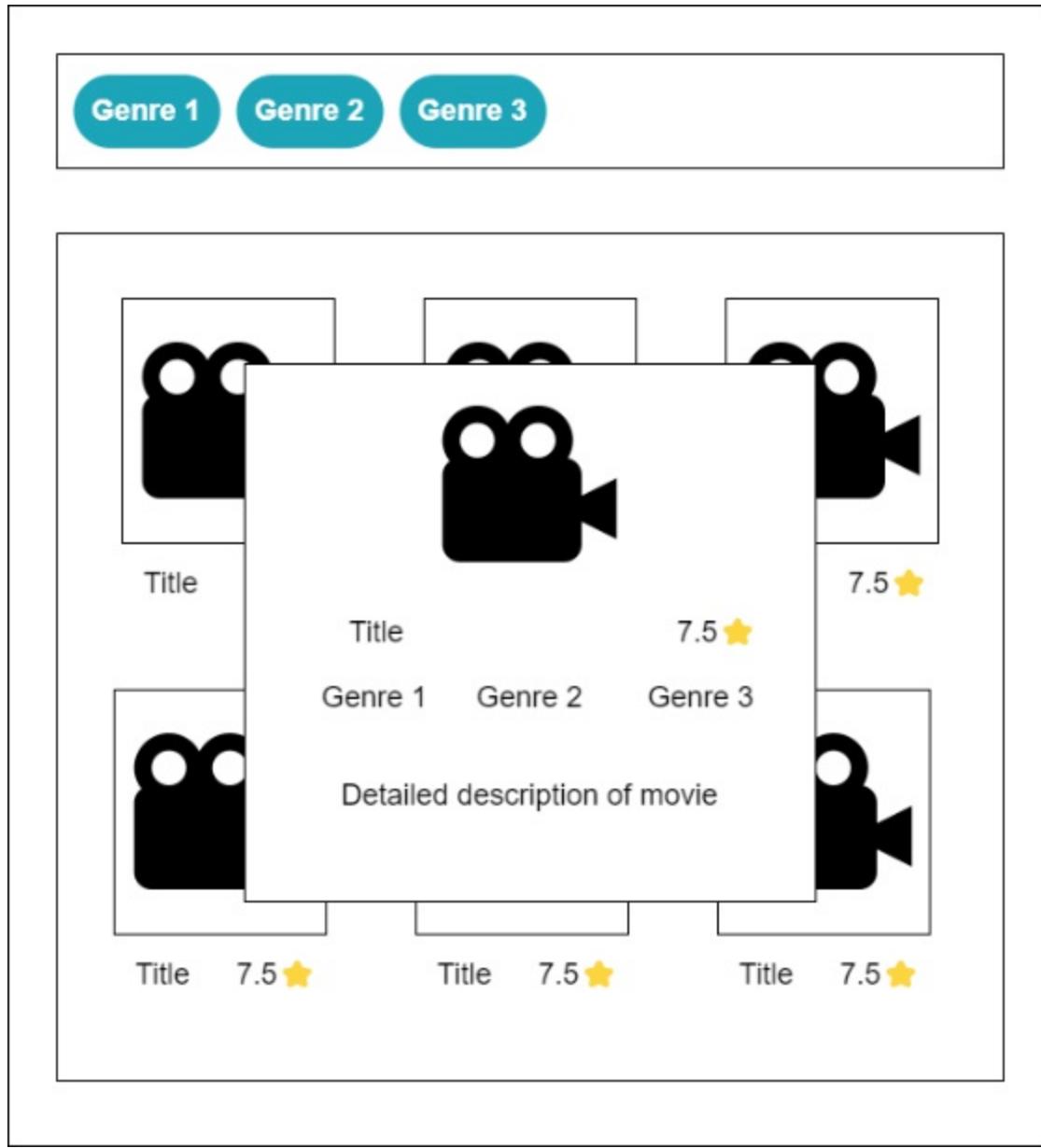


Figure 6.6 The user can tap or click on one of the movies to see more information about it. A popup will show the poster, title and rating, but also the genres the movie fits into and the detailed description or blurb.



We're going to use `BindableLayout` with two `FlexLayouts` in `MauiMovies`: one for the genre list, and one for the movie list.

6.2.1 Creating the MauiMovies MainPage

We'll build this app using the free API available from <https://www.themoviedb.org>. Head over there now and sign up. From within the account settings page, you can request an API key. Fill out the details to request an API key; we'll use it in the `MauiMovies` app.

Create a new .NET MAUI app from the blankmaui template and call it MauiMovies.

Before we can start building the UI, we'll need to add some types to deserialise the data we get back from the API. If you're using Visual Studio, you can copy JSON to your clipboard and use the Edit | Paste Special menu to have it automatically converted to C# classes. If not, you can use <https://json2csharp.com>. In any case, I've already done this, so you don't need to.

First, create a folder in the project called `Models`, and in here add a class called `MovieResult`. Listing 6.2 shows the code for this class.

Listing 6.2 The `MovieResult` class

```
namespace MauiMovies;

public class MovieResult
{
    public string original_language { get; set; }
    public string original_title { get; set; }
    public string poster_path { get; set; }
    public bool video { get; set; }
    public double vote_average { get; set; }
    public string overview { get; set; }
    public string release_date { get; set; }
    public int vote_count { get; set; }
    public int id { get; set; }
    public bool adult { get; set; }
    public string backdrop_path { get; set; }
    public string title { get; set; }
    public List<int> genre_ids { get; set; }
    public double popularity { get; set; }
    public string media_type { get; set; }
}
```

Next, add a class to the `Models` folder called `TrendingMovies`. Listing 6.3 shows the code for this class.

Listing 6.3 The `TrendingMovies` class

```
namespace MauiMovies;
```

```
public class TrendingMovies
{
    public int page { get; set; }
    public List<MovieResult> results { get; set; }
    public int total_pages { get; set; }
    public int total_results { get; set; }
}
```

Add another class to the `Models` folder called `Genre`. Listing 6.4 shows the code for this class.

Listing 6.4 The Genre class

```
namespace MauiMovies;

public class Genre
{
    public int id { get; set; }
    public string name { get; set; }
}
```

Next, add a class to the `Models` folder called `GenreList`. Listing 6.5 shows the code for this class.

Listing 6.5 The GenreList class

```
namespace MauiMovies;

public class GenreList
{
    public List<Genre> genres { get; set; }
}
```

The last class we need to add to the `Models` folder is `UserGenre`. This will subclass the `Genre` class but add a `Selected` property. Listing 6.6 shows the code for this class.

Listing 6.6 The UserGenre class

```
namespace MauiMovies;

public class UserGenre : Genre
```

```
{  
    public bool Selected { get; set; }  
}
```

Now that we've got all the types we need in place, let's add the functionality to get the data we want from the API. Open the `MainPage.xaml` file in the `MauiMovies` app and **delete everything between the** `<ContentPage...>...` `</ContentPage>` **tags**. Next, open `MainPage.xaml.cs`, and **delete the count and the OnCounterClicked method**.

Now we're ready to build the functionality to get movies and genres from the API. The following steps will set us up to build the UI with this data.

1. Add a field for the API key
2. Add a field for the base URI for the API
3. Add a field for the top 20 trending movies from the API
4. Add a field for the list of genres from the API
5. Add a field for the HttpClient
6. Add a loading property to show an ActivityIndicator while we wait for the data
7. Add a property for the genres the user has selected
8. Add a property for the movies filtered by genre

Listing 6.7 shows the updated `MainPage.xaml.cs` with the added fields and properties.

Listing 6.7 The updated MainPage.xaml.cs with the fields and properties

```
using System.Collections.ObjectModel;  
using System.Net.Http.Json;  
  
namespace MauiMovies;  
  
public partial class MainPage : ContentPage  
{  
    string _apiKey = "[your API key]";  
    string _baseUri = "https://api.themoviedb.org/3";  
  
    private TrendingMovies _movieList;  
  
    private GenreList _genres;
```

```

public ObservableCollection<Genre> Genres { get; set; } = new
public ObservableCollection<MovieResult> Movies { get; set; }
public bool IsLoading { get; set; }
private readonly HttpClient _httpClient;
public MainPage()
{
    InitializeComponent();
}
}

```

Next, we need to update the page's constructor. We'll set the page's binding context to itself, so that the UI can bind to the properties we just created, and we'll instantiate the `HttpClient` with the API's base URI. Listing 6.8 shows the updated constructor in `MainPage.xaml.cs` (with the additions in **bold**).

Listing 6.8 The MainPage.xaml.cs constructor

```

public MainPage()
{
    InitializeComponent();
    BindingContext = this;
    _httpClient = new HttpClient { BaseAddress = new Uri(_bas
}

```

The last step before we start building the UI is to do the initial data load. Override the page's `OnAppearing` method and make it `async`. In here. Use the `HttpClient` to retrieve and deserialise the movies and genres from the API. Listing 6.9 shows the overridden `OnAppearing` method.

Listing 6.9 The MainPage.xaml.cs OnAppearing method

```

protected override async void OnAppearing()
{
    base.OnAppearing();
    IsLoading = true;
    OnPropertyChanged(nameof(IsLoading));

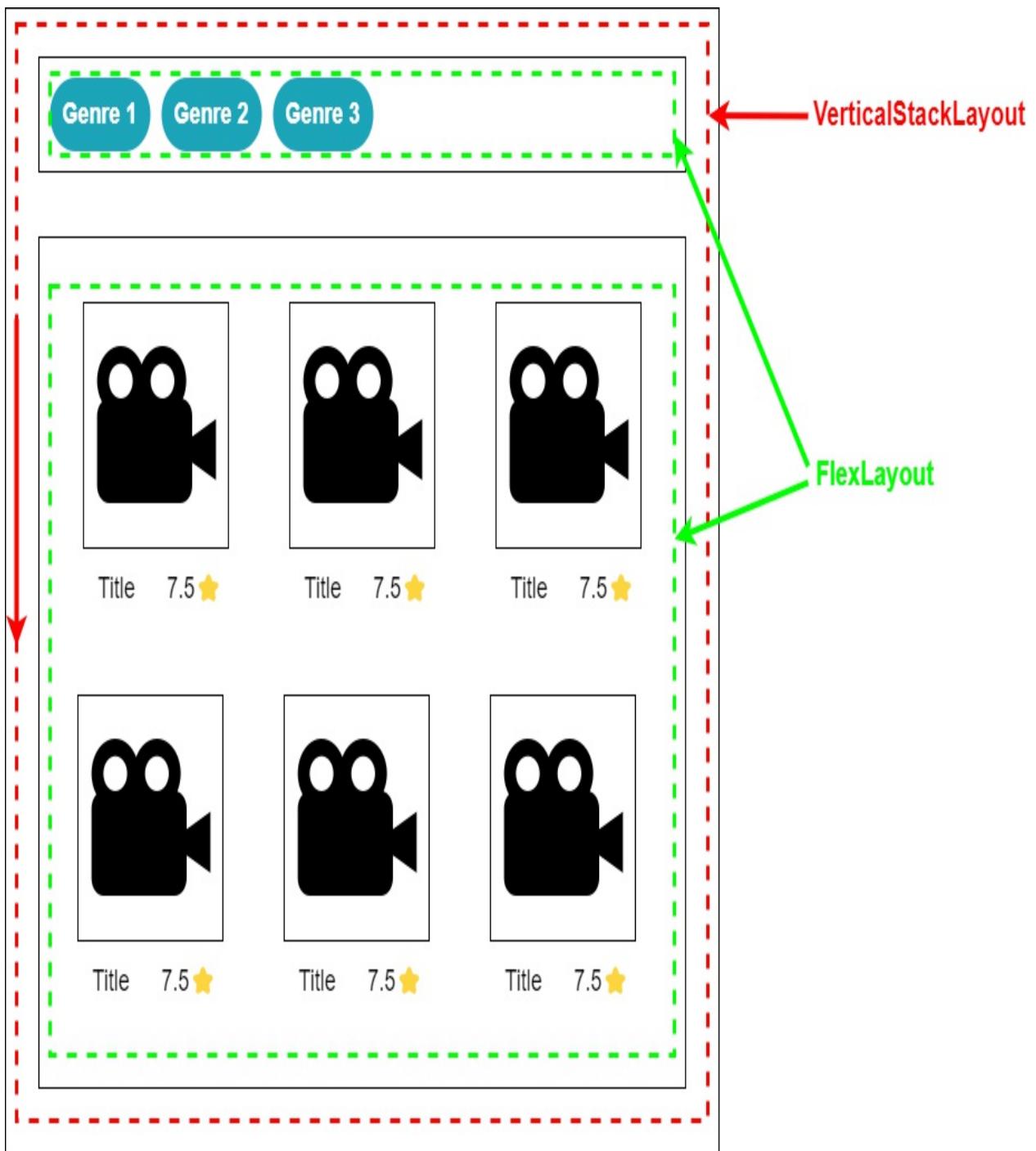
    _genres = await _httpClient.GetFromJsonAsync<GenreList>($

```

```
_movieList = await _httpClient.GetFromJsonAsync<TrendingM  
IsLoading = false;  
 OnPropertyChanged(nameof(IsLoading));  
}
```

Now that we've got the initialisation code finished, we can start laying out the structure for the UI. Figure 6.7 shows the breakdown of the layouts we'll use for MauiMovies.

Figure 6.7 The main layout for the page is a `VerticalStackLayout`. This will arrange the child views vertically from top to bottom. The first view to add to the `VerticalStackLayout` is a `FlexLayout` that will show the selected genres. After that we'll add another `FlexLayout` to show the filtered list of movies.



In the `MainPage.xaml` file, add `VerticalStackLayout` with Spacing of 10 and Padding of 30. The first item to add to the `VerticalStackLayout` is the genre selection box. To do this, we'll add another `VerticalStackLayout` that will have a `Label` that says “Genres”, and a `FlexLayout` underneath it showing the selected genres. We’ll use `BindableLayout` to provide the data source for the selected genres, and these will be presented as ‘chips’ or pills.

We can use Border for two things: to wrap the whole genre selection box in a nice border and to provide the chip effect. Finally, we will add a gesture recogniser, so that when the user taps on the genre selection box, we can present a popup showing the list of available genres for them to choose from. Listing 6.10 shows the updated `MainPage.xaml`, with the added genre selection box.

Listing 6.10 MainPage.xaml with the genre selection box

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

    <VerticalStackLayout Spacing="10" Padding="30">

        <Border Stroke="Black"
            StrokeThickness="2"
            StrokeShape="RoundRectangle 5">
            <VerticalStackLayout Padding="20">
                <VerticalStackLayout.GestureRecognizers>
                    <TapGestureRecognizer NumberOfTapsRequired="1"
                        Command="{Binding Choos
                </VerticalStackLayout.GestureRecognizers>
                <Label Text="Genres:"/>
                <FlexLayout BindableLayout.ItemsSource="{Binding
                    JustifyContent="SpaceEvenly"
                    Wrap="Wrap">
                    <BindableLayout.ItemTemplate>
                        <DataTemplate>
                            <Border Stroke="CadetBlue"
                                StrokeShape="RoundRectangle 1
                                <Label Text="{Binding name}"
                                    TextColor="White"
                                    Padding="10"
                                    BackgroundColor="CadetBlue
                                        </Border>
                                </DataTemplate>
                            </BindableLayout.ItemTemplate>
                        </FlexLayout>
                    </VerticalStackLayout>
                </Border>
            </VerticalStackLayout>
        </ContentPage>
```

Now that we've got the genres fixed up let's add the layout for the actual movies. In `MainPage.xaml`, underneath the `Border` we just added in the `VerticalStackLayout`, add a `ScrollView`. Inside this, add a `FlexLayout`, make it a `BindableLayout` and bind the `ItemsSource` to the `Movies` collection. Set the `JustifyContent` property to `SpaceBetween` and the `Wrap` property to `Wrap`.

Listing 6.11 shows the code to add to `MainPage.xaml`, with the added code in **bold**.

Listing 6.11 The movies layout to add to MainPage.xaml

```
<?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="MauiMovies.MainPage">

    <VerticalStackLayout Spacing="10" Padding="30">

        ... [Code omitted for clarity]...

        <ScrollView>
            <FlexLayout BindableLayout.ItemsSource="{Binding Movies}"
                JustifyContent="SpaceEvenly"
                Wrap="Wrap">
                <BindableLayout.ItemTemplate>
                    <DataTemplate>
                        </DataTemplate>
                </BindableLayout.ItemTemplate>
            </FlexLayout>
        </ScrollView>
    </VerticalStackLayout>
</ContentPage>
```

For the `DataTemplate`, we'll use a `VerticalStackLayout` to arrange the poster, title and rating. The title and rating can go into a `HorizontalStackLayout` underneath the poster, and the poster should have a shadow to add some depth to the UI.

Listing 6.12 shows the code to add to `MainPage.xaml`, with the added code in **bold**.

Listing 6.12 The movies layout to add to MainPage.xaml

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

    <VerticalStackLayout Spacing="10" Padding="30">

        ... [Code omitted for clarity]...

        <ScrollView>
            <FlexLayout BindableLayout.ItemsSource="{Binding Movi
                JustifyContent="SpaceEvenly"
                Wrap="Wrap">
                <BindableLayout.ItemTemplate>
                    <DataTemplate>
                        <VerticalStackLayout WidthRequest="200"
                            Margin="30"
                            Spacing="10">
                            <VerticalStackLayout.GestureRecognize
                                <TapGestureRecognizer NumberOfTap
                                    Command="{B
                            </VerticalStackLayout.GestureRecogniz

                            <Image Source="{Binding poster_path}"
                                Aspect="AspectFit">
                                <Image.Shadow>
                                    <Shadow Brush="Black"
                                        Opacity="0.6"
                                        Offset="5,5"
                                        Radius="20"/>
                                </Image.Shadow>
                            </Image>
                            <HorizontalStackLayout>
                                <Label Text="{Binding title}"
                                    FontAttributes="Bold"
                                    WidthRequest="150"
                                    LineBreakMode="TailTruncat
                                <Label HorizontalOptions="EndAndE
                                    HorizontalTextAlignment="E
                                    WidthRequest="50"
                                    Text="{Binding vote_averag
                            </HorizontalStackLayout>
                        </VerticalStackLayout>
                    </DataTemplate>
                </BindableLayout.ItemTemplate>
```

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

The main UI is now complete, and we're almost ready to run MauiMovies and see it in action. But first we need to add a little bit more code. The poster_path property that we get from the API is only a partial URL, so we'll need to hydrate that with a base URL for the images. We'll add a field for this base URL and hydrate the poster_path when we load the movies from the API. I got this URL from the API's /Configuration endpoint which is covered in their documentation.

We also need to add a method to update the `Movies ObservableCollection` with the filtered movie list. When the page loads this will be all movies as the user won't have chosen any genres yet. We also added bindings in the UI to two `Commands`, so we need to add these too (although we don't need to do anything with them yet).

Listing 6.13 shows the updated `MainPage.xaml.cs` with the added code shown in **bold**.

Listing 6.13 MainPage.xaml.cs with the logic to load movies to the UI

```
using CommunityToolkit.Maui.Views;
using System.Collections.ObjectModel;
using System.Net.Http.Json;
using System.Windows.Input;

namespace MauiMovies;

public partial class MainPage : ContentPage
{
    string _apiKey = "[YOUR API KEY HERE]]";
    string _baseUri = "https://api.themoviedb.org/3/";

    string _imageBaseUrl = "https://image.tmdb.org/t/p/w500";

    TrendingMovies _movieList;

    GenreList _genres;

    public ObservableCollection<UserGenre> Genres { get; set; } =
```

```
public ObservableCollection<MovieResult> Movies { get; set; }

public ICommand ChooseGenres { get; set; }

public ICommand ShowMovie { get; set; }

public bool IsLoading { get; set; }

HttpClient _httpClient;

List<UserGenre> _genreList { get; set; } = new();

public MainPage()
{
    InitializeComponent();
    BindingContext = this;
    _httpClient = new HttpClient { BaseAddress = new Uri(_bas
}

protected override async void OnAppearing()
{
    base.OnAppearing();
    IsLoading = true;
    OnPropertyChanged(nameof(IsLoading));

    _genres = await _httpClient.GetFromJsonAsync<GenreList>($
    _movieList = await _httpClient.GetFromJsonAsync<TrendingM

    foreach (var movie in _movieList.results)
    {
        movie.poster_path = $"{_imageBaseUrl}{movie.poster_pa
    }

    foreach (var genre in _genres.genres)
    {
        _genreList.Add(new UserGenre
        {
            id = genre.id,
            name = genre.name,
            Selected = false
        });
    }
}

LoadFilteredMovies();
```

```

        IsLoading = false;
        OnPropertyChanged(nameof(IsLoading));
    }

private void LoadFilteredMovies()
{
    Movies.Clear();

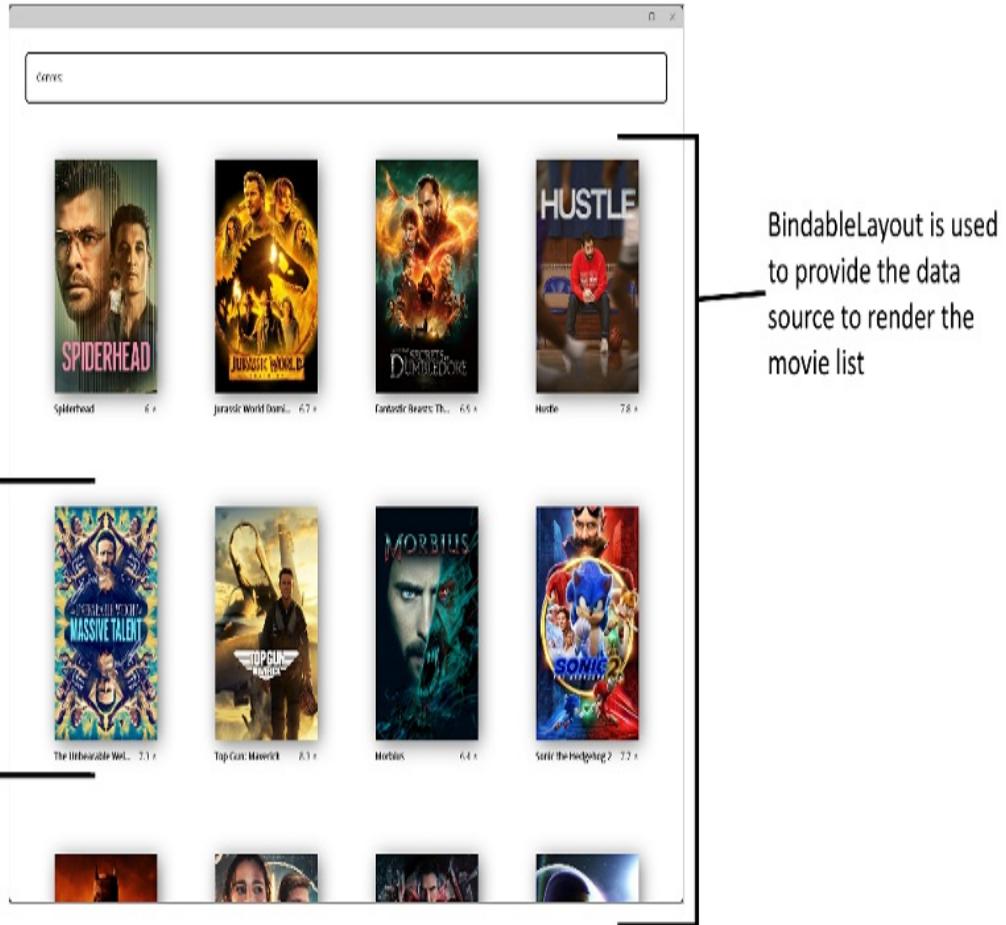
    if (_genreList.Any(g => g.Selected))
    {
        var selectedGenreIds = _genreList.Where(g => g.Select

            foreach (var movie in _movieList.results)
            {
                if (movie.genre_ids.Any(id => selectedGenreIds.Co
                {
                    Movies.Add(movie);
                }
            }
        }
    }
    else
    {
        foreach(var movie in _movieList.results)
        {
            Movies.Add(movie);
        }
    }
}
}

```

We've now got all the code we need to get a list of movies from themoviedb.org and display them in our UI (although we've still got the filtering and movie details to go). Run MauiMovies, and you should see something like figure 6.8.

Figure 6.8 MauiMovies running on Windows. FlexLayout is used to arrange the movies and wrap each movie onto the next line if it doesn't fit. The movies come from an ObservableCollection, and BindableLayout is used to bind that collection to the FlexLayout. Each movie is rendered based on a template that includes the poster, title, and rating.



With the app running we can now see the list of movies from the API. We've got two things left to do: filter based on genre and show details for a movie. Even though we can't see it yet, the genre box at the top of the screen is already set up to show a list of chips with the names of the genres the user has chosen for filtering, we just need to give the user a way to choose them.

We're going to use popups for both of these (the genre list and movie details). In the next section, we'll see how to create the genre list and movie details popups.

6.2.2 Creating the Popup Pages

We'll use popups to display genre selection list and the movie details, and the .NET MAUI Community Toolkit has a nice popup view that we can use for these.

The first step is to install the `CommunityToolkit.Maui` NuGet package into the `MauiMovies` project.

Now we need to add the popup pages; let's start with the genre list. The easiest way to create these pages is to add a .NET MAUI ContentPage (using the template) and modify it to suit our needs. Add a new page using the .NET MAUI ContentPage (XAML) template and call it `GenreListPopup`.

Open the `GenreListPopup.xaml.cs` file and change the inherited type from `ContentPage` to `Popup`. You'll need to bring in the `CommunityToolkit.Maui.Views` namespace. You'll see some errors now because of a mismatch with the XAML, so switch over to the `GenreListPopup.xaml` to fix it up.

First, we need to bring in the .NET MAUI Community Toolkit XAML namespace, and we'll assign it to the name `mct`. Then, we can change the type from `ContentPage` to `mct:Popup`. You'll get one more error because the `ContentPage` template includes a `Title` property which `Popup` doesn't have, so delete the `Title` property. `Popup` also lets us define the size in XAML, so we can add this in the opening tag. Finally, delete all the content from the template (the `VerticalStackLayout` and its children), and then you should be all clear of errors and ready to start adding some views.

Listing 6.14 shows the code for the `GenreListPopup.xaml.cs` file.

Listing 6.14 `GenreListPopup.xaml.cs`

```
using CommunityToolkit.Maui.Views; #A  
namespace MauiMovies;  
  
public partial class GenreListPopup : Popup #B  
{  
    public GenreListPopup()  
    {  
        InitializeComponent();  
    }  
}
```

Listing 6.15 shows the code for the `GenreListPopup.xaml` file.

Listing 6.15 GenreListPopup.xaml

```
<?xml version="1.0" encoding="utf-8" ?>
<mct:Popup xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui"
    Size="600,600"
    x:Class="MauiMovies.GenreListPopup">
</mct:Popup>
```

With the basic Popup created, let's add some code to handle the genre list that we'll pass through from the main page. The first thing we'll need is to accept the list of genres for the user to select, so update the constructor to take a list of type `UserGenre`. We'll need to add an `ObservableCollection` to hold the `UserGenres` that we receive and to bind to in the UI.

To use that binding, we also need to set the `Popup`'s binding context to itself.

Listing 6.16 The ObservableCollection and Constructor

```
using CommunityToolkit.Maui.Views;
using System.Collections.ObjectModel;

namespace MauiMovies;

public partial class GenreListPopup : Popup
{
    public ObservableCollection<UserGenre> Genres { get; set; }

    public GenreListPopup(List<UserGenre> Genres)
    {
        BindingContext = this;
        this.Genres = new ObservableCollection<UserGenre>(Genres);
        InitializeComponent();
    }
}
```

We will need two event handlers in the code. The first one will handle when a selection changes in the genre list. We can use the event args passed to the event handler to get the currently selected items and iterate through the `Genres` collection and for each entry, check to see if it is in the selected items.

If it is, we mark its `Selected` property as `true`. Note that this is something we can do with `CollectionView` but wouldn't be able to do with `BindableLayout` attached to another layout. We'll also add a field that can be used to track when the user has made a selection change.

Listing 6.17 The CollectionView_SelectionChanged method

```
private bool _selectionHasChanged = false;

private void CollectionView_SelectionChanged(object sender,
{
    selectionHasChanged = true;

    var selectedItems = e.CurrentSelection;

    foreach (var genre in Genres)
    {
        if (selectedItems.Contains(genre))
        {
            genre.Selected = true;
        }
        else
        {
            genre.Selected = false;
        }
    }
}
```

The second event handler will dismiss the popup when the user clicks on a button. The popup base class has a `Close()` method we can use, so we'll just call this. We can use this method to return a value to our page that called the popup, so that it knows whether the user has made a selection change. We'll initialise a field with a value of `false` and change it to `true` if the selection changed event handler is called.

We use a property inherited from `Popup` called `ResultWhenUserTapsOutsideOfPopup` to also return the value of this field when the user to dismisses the popup by tapping outside rather than using the Confirm button (which we will add shortly). This will need to be set in the constructor.

Listing 6.18 The logic for indicating if the selection has changed

```

using CommunityToolkit.Maui.Views;
using System.Collections.ObjectModel;

namespace MauiMovies;

public partial class GenreListPopup : Popup
{
    // remaining code omitted

    Public GenreListPopup(List<UserGenre> Genres)
    {
        // remaining code omitted

        ResultWhenUserTapsOutsideOfPopup = _selectionHasChanged;
    }

    // remaining code omitted

    private void Button_Clicked(object sender, EventArgs e) =
}

```

That's all the logic that this popup, so let's add the UI now. In the `GenreListPopup.xaml` file, we'll add a `VerticalStackLayout` to arrange all the views. We'll add a `Confirm` `Button` to the top, and then a `CollectionView` to show the available list of genres.

`Listing 6.19` shows the complete code for `GenreListPopup.xaml`.

Listing 6.19 `GenreListPopup.xaml`

```

<?xml version="1.0" encoding="utf-8" ?>
<mct:Popup xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui"
            x:Class="MauiMovies.GenreListPopup">
    <VerticalStackLayout Spacing="10"
                        Padding="10">

        <Button Text="Confirm"
                Clicked="Button_Clicked"/>

        <CollectionView ItemsSource="{Binding Genres}"
                       SelectionMode="Multiple"
                       SelectionChanged="CollectionView_Selectio
        <CollectionView.ItemTemplate>

```

```

        <DataTemplate>
            <Label Text="{Binding name}" />
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

</VerticalStackLayout>
</mct:Popup>

```

We've now got a complete popup for displaying the list of genres and letting the user select one or more genres. As the genres are passed by reference, the user's selection here will be reflected in the `MainPage` when we go back to it.

The last piece of UI to add for the `MauiMovies` app is another popup to show movie details. Use the same process to add another popup called `MovieDetailsPopup` (add a `ContentPage` from the template, update the base class, bring in the `Community Toolkit` namespace and change the type in XAML).

We'll need properties for all the key details about the movie: title, description, poster URL, a list of genres, and the rating. We can add a `MovieResult` as a constructor parameter and assign most of these values based on this parameter. `MovieResult` only contains a list of genre IDs though, so we should pass in the list of genres too and assign relevant names from here based on the IDs of the genres that apply to the movie.

Finally, we can set the popup's binding context to itself. Listing 6.20 shows the complete code for `MovieDetailsPopup.xaml.cs`.

Listing 6.20 MovieDetailsPopup.xaml.cs

```

using CommunityToolkit.Maui.Views;

namespace MauiMovies;

public partial class MovieDetailsPopup : Popup
{
    public string Title { get; set; }

    public string Description { get; set; }

    public string PosterUrl { get; set; }

```

```

public List<string> Genres { get; set; } = new();

public double Rating { get; set; }

public MovieDetailsPopup(MovieResult movie, List<Genre> g
{
    Size = new Size(600, 600);

    Title = movie.title;

    Description = movie.overview;

    PosterUrl = movie.poster_path;

    Rating = movie.vote_average;

    foreach (var id in movie.genre_ids)
    {
        Genres.Add(genres.Where(g => g.id == id).Select(g => g.name).First());
    }

    BindingContext = this;
}

InitializeComponent();
}
}

```

The movie details popup now has all the code it needs to display details about a movie that gets passed into it, so let's create the UI. We'll use a `VerticalStackLayout` to arrange the views. At the top we'll use an `Image` to show the poster. After that we can use a `FlexLayout` to show the title and rating. After that, another `FlexLayout` can show the genres and at the end a `Label` can show the description.

Listing 6.21 shows the code for `MovieDetailsPopup.xaml`.

Listing 6.21 MovieDetailsPopup.xaml

```

<?xml version="1.0" encoding="utf-8" ?>
<mct:Popup xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:mct="http://schemas.microsoft.com/dotnet/2022/xaml/mct"
            x:Class="MauiMovies.MovieDetailsPopup">
    <VerticalStackLayout Padding="20"

```

```

        Spacing="20">

    <Image Source="{Binding PosterUrl}"
           HeightRequest="300"
           Aspect="AspectFit"/>

    <FlexLayout JustifyContent="SpaceBetween">
        <Label Text="{Binding Title}"
               FontAttributes="Bold"/>
        <Label HorizontalTextAlignment="End"
               Text="{Binding Rating, StringFormat='{0} ☆ {1}'}"
        </FlexLayout>

    <FlexLayout BindableLayout.ItemsSource="{Binding Genres}"
               JustifyContent="SpaceEvenly"
               Wrap="Wrap">
        <Label Text="{Binding .}"/>
    </FlexLayout>

    <Label Text="{Binding Description}" />
</VerticalStackLayout>
</mct:Popup>
```

This gives us all the UI components for the MauiMovies app, all that's left to do is wire them all up to the main page.

6.2.3 Showing the Popup Pages

Let's go back to the main page and wire up some logic to show our new popup. In `MainPage.xaml.cs`, add a method with a return type of `Task` called `ShowGenreList`, and make it `async`. In this method, instantiate a new `GenreListPopup`, passing the `_genreList` field to its constructor. We can show the popup by calling an extension method called `ShowPopupAsync`. We'll await the result, and then ensure that the result is true (which it will be if the user has made a selection change before closing the popup).

If the result is true, we'll clear the `Genres ObservableCollection`, then repopulate with the updated `_genreList`. Then we can call the `LoadFilteredMovies` method which will update the UI to show movies that meet the users selected genre criteria. Now that this method exists, we can wire up the `ChooseGenres ICommand` to call it, using an expression body.

Listing 6.22 shows the `MainPage.xaml.cs` file, but only with the added code. The remaining code has been omitted for brevity.

Listing 6.22 The new code in MainPage.xaml.cs

```
using CommunityToolkit.Maui.Views;
using System.Collections.ObjectModel;
using System.Net.Http.Json;
using System.Windows.Input;

namespace MauiMovies;

public partial class MainPage : ContentPage
{
    // ...

    public ICommand ChooseGenres => new Command(async () => await
    // ...

    private async Task ShowGenreList()
    {
        var genrePopup = new GenreListPopup(_genreList);

        var selected = await this.ShowPopupAsync(genrePopup);

        if ((bool)selected)
        {
            Genres.Clear();
            foreach(var genre in _genreList)
            {
                if (genre.Selected)
                {
                    Genres.Add(new Genre
                    {
                        name = genre.name
                    });
                }
            }

            LoadFilteredMovies();
        }
    }
}
```

We've now got all the code we need to filter the movie list based on the

user's selection. Run the app and experiment with different genre selections. You should see results similar to figures 6.9 and 6.10.

Figure 6.9 MauiMovies running on Windows showing popup that is displayed by the ShowPopupAsync method. A CollectionView shows the list of genres, and selecting one or more triggers the SelectionChanged event. Dismissing the popup, either by tapping Confirm or tapping on the background outside the popup, returns the _selectionChanged value.

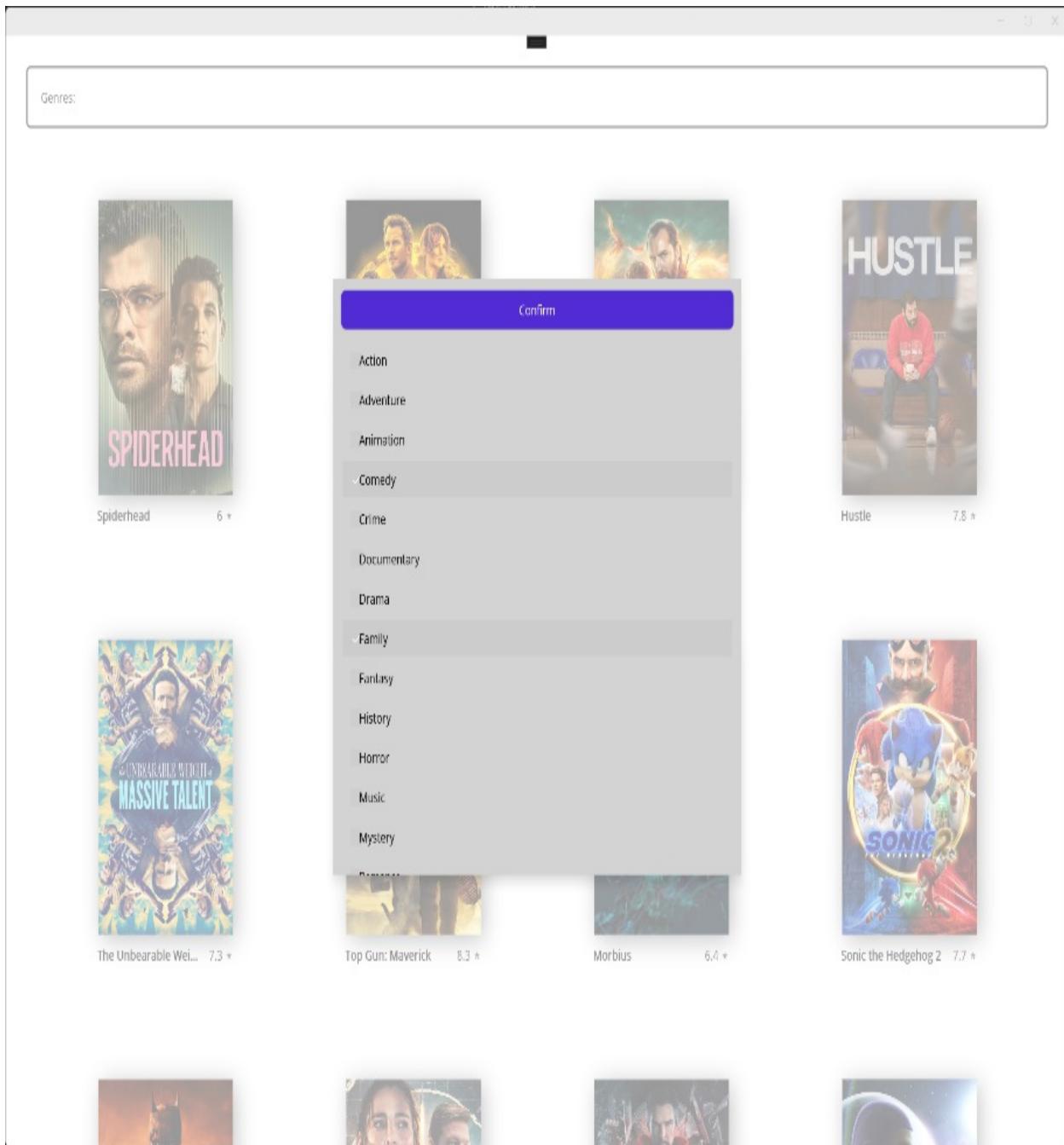
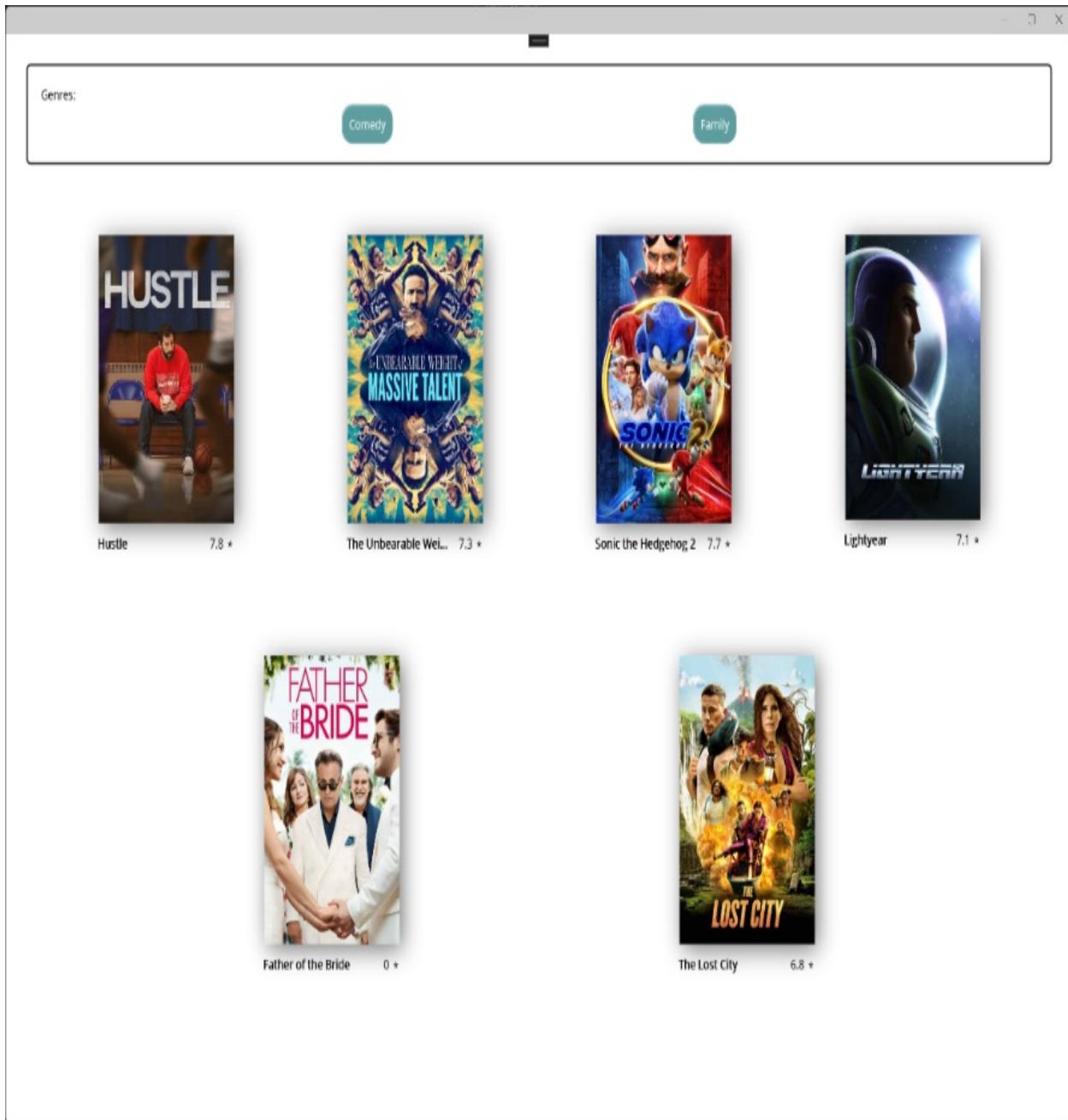


Figure 6.10 MauiMovies running on Windows. The user's genre selections are shown in the Genres box, and only movies meeting those criteria are shown.



The last thing we need to do now is show the movie details popup when the user taps or clicks on a movie. We'll need a method we can call that will show a `MovieDetailsPopup` with the selected movie and the list of genres passed into its constructor, and we'll need to wire up the `ShowMovie` `ICommand` we already added to this method. The method and the `ICommand`

will both need to be take the `MovieResult` type.

Listing 6.23 shows these updates to `MainPage.xaml.cs`. Only the new or changed code is shown; the rest is omitted for brevity.

Listing 6.23 The final changes to `MainPage.xaml.cs`

```
using CommunityToolkit.Maui.Views;
using System.Collections.ObjectModel;
using System.Net.Http.Json;
using System.Windows.Input;

namespace MauiMovies;

public partial class MainPage : ContentPage
{
    ...

    public ICommand ShowMovie => new Command<MovieResult>((movie)

        ...

        private void ShowMovieDetails(MovieResult movie)
        {
            var moviePopup = new MovieDetailsPopup(movie, _genres.gen
                this.ShowPopup(moviePopup);
        }
}
```

Lastly, we'll need to update the `TapGestureRecognizer` attached to the `VerticalStackLayout` for each movie. The binding we've set for the `Command` property won't work; we've set the binding to the `ShowMovie` method, but the `ShowMovie` method exists on the `MainPage` class, *not* the `MovieResult` class which is the binding context for the item in the bindable layout.

We can fix this by using the `source` markup extension and setting a reference to the page. As we've now typed the `Command` to a `MovieResult`, which we also need for the `ShowMovieDetails` method, we will need a `CommandParameter`. This one is easy though, as we can just set it to the binding for the item itself.

Listing 6.24 shows the changes to the `MainPage.xaml` file. Most of the unchanged code is omitted for brevity, and the changed code is shown in **bold**.

Listing 6.24 The final changes to MainPage.xaml

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

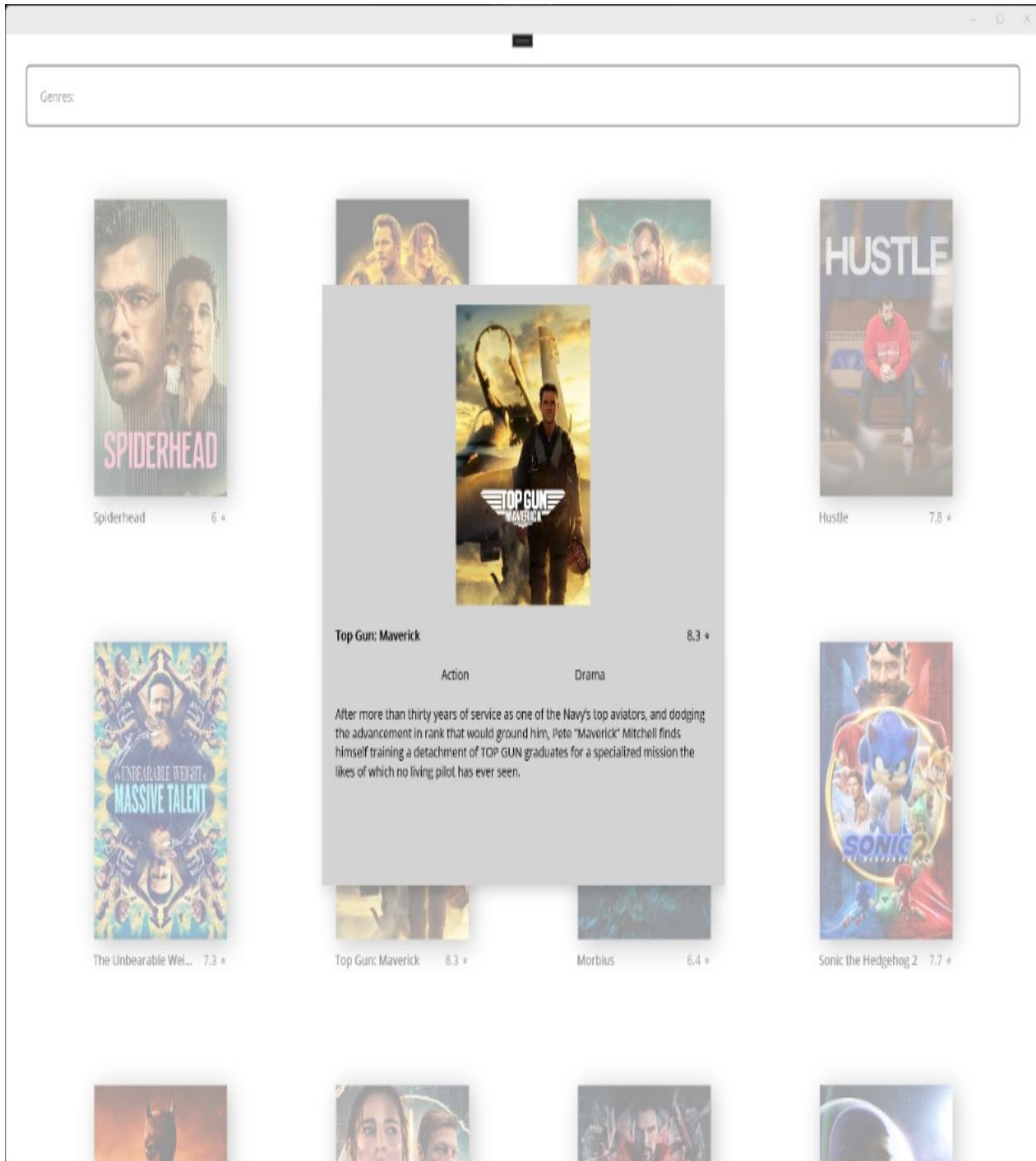
    <VerticalStackLayout Spacing="10" Padding="30">

        ...

        <ScrollView>
            <FlexLayout BindableLayout.ItemsSource="{Binding Movi
                JustifyContent="SpaceEvenly"
                Wrap="Wrap">
                <BindableLayout.ItemTemplate>
                    <DataTemplate>
                        <VerticalStackLayout WidthRequest="200"
                            Margin="30"
                            Spacing="10">
                            <VerticalStackLayout.GestureRecognize
                                <TapGestureRecognizer NumberOfTap
                                    Command="{B
                                    CommandPara
                            </VerticalStackLayout.GestureRecogniz
                        ...
                    </DataTemplate>
                </BindableLayout.ItemTemplate>
            </FlexLayout>
        </ScrollView>
    </ContentPage>
```

This adds the functionality we need to tap or click on a movie to see the movie details. If you run MauiMovies now and tap on a move, you should see something like figure 6.11.

Figure 6.11 MauiMovies running on Windows. The user has tapped on one of the movies to bring up the MovieDetailsPopup, which shows a bit more information about the selected movie.

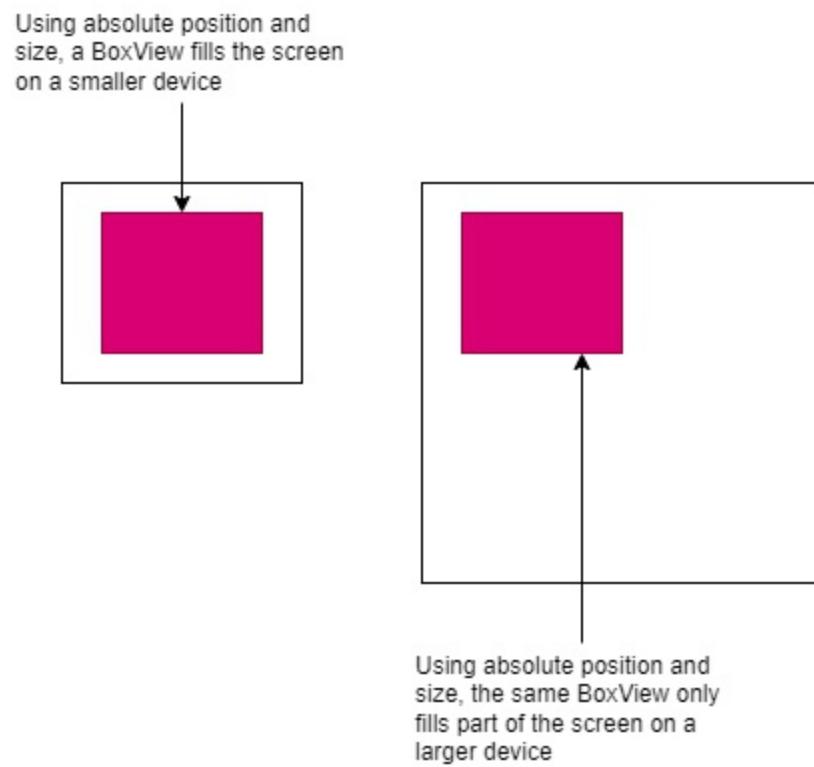


This completes the MauiMovies app! Run the app now and click around, experimenting with changing genres and viewing movie details.

6.3 Absolute Layout

`AbsoluteLayout` lets you use explicit values to position views on the screen. This can be useful, but you have to be cautious with it; everyone's screen will be a different height and width so even using explicit values, you can't guarantee the absolute position of a view.

Figure 6.12 Using `AbsoluteLayout` can have unintended consequences. In this example a `BoxView` has been added with an absolute position and absolute size. It takes up nearly the whole screen on a smaller device, but just a small part of the top-left corner on a bigger screen.

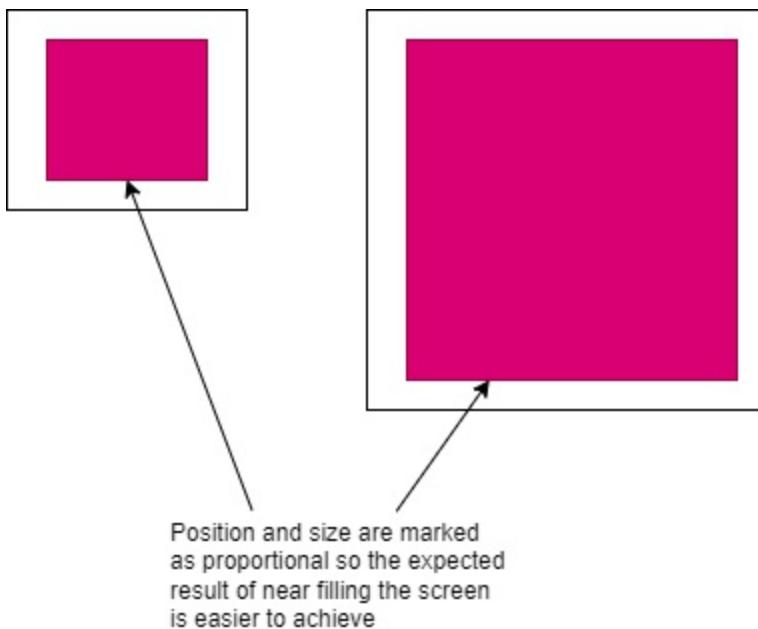


`AbsoluteLayout` uses two attached properties, called `LayoutParams` and `LayoutFlags`, so that a view can position itself within an `AbsoluteLayout`. The `LayoutParams` is of type `Rect`, meaning it has four properties: `x`, `y`, `width`, and `height`.

The first two properties specify the top left position of the view within the `AbsoluteLayout`, and the second two define the size of the view. These properties have default values of `0, 0, Auto, Auto`, meaning that if you add any view as a child to an `AbsoluteLayout` you can omit the `LayoutParams`, and the child view will be positioned in the top left and will size automatically to accommodate its contents.

Despite the name, `AbsoluteLayout` becomes more useful when you use it for proportional sizing and positioning. The `LayoutFlags` property is an enum that lets you specify which of the `LayoutBounds` are proportional and which are absolute. It provides a lot of flexibility, letting you declare whether x, y or both (position) are proportional, height, width or both (size) are proportional, whether all values are proportional, or none are.

Figure 6.13 In this example, the position and size of the `BoxView` are proportional to the `AbsoluteLayout`. It's easier to achieve a desired effect (in this case, having a `BoxView` that fills most of the screen) using proportional size and position.



To help make sense of this let's see a simple example. Create a new .NET MAUI app from the `blankmaui` template and call it `MauiFab`. Wrap the entire contents of the `MainPage` in an `AbsoluteLayout`. Listing 6.25 shows what this should look like, with the added parts in **bold**. The contents of the `ScrollView` have been omitted for brevity.

Listing 6.25 `MainPage.xaml` in `MauiFab`

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

```

<AbsoluteLayout>          #A
    <ScrollView>
        ...
    </ScrollView>
</AbsoluteLayout>
</ContentPage>

```

We're going to add a simple floating action button (FAB) to this page. To do this, we'll add a `Button` with a height and width of 100, and a corner radius of 50 to make it circular. We'll set the `LayoutFlags` to `PositionProportional`, so that we can ensure it always appears in the same relative position, and set the `LayoutBounds` to put the x and y both at 0.9. As the position is proportional, 0.9 is 9/10 of the way across (for x) and down (for y), so it will be positioned in the bottom right.

Listing 6.26 shows the `MainPage.xaml` with the `Button` added in **bold**. The contents of the `ScrollView` have been omitted for brevity.

Listing 6.26 MainPage.xaml with the FAB added

```

<?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="FabMaui.MainPage">

    <AbsoluteLayout>
        <ScrollView>
            ...
        </ScrollView>

        <Button CornerRadius="50"
               Text="Fab!"
               FontSize="Large"
               AbsoluteLayout.LayoutBounds="0.9,0.9,100,100"
               AbsoluteLayout.LayoutFlags="PositionProportional"

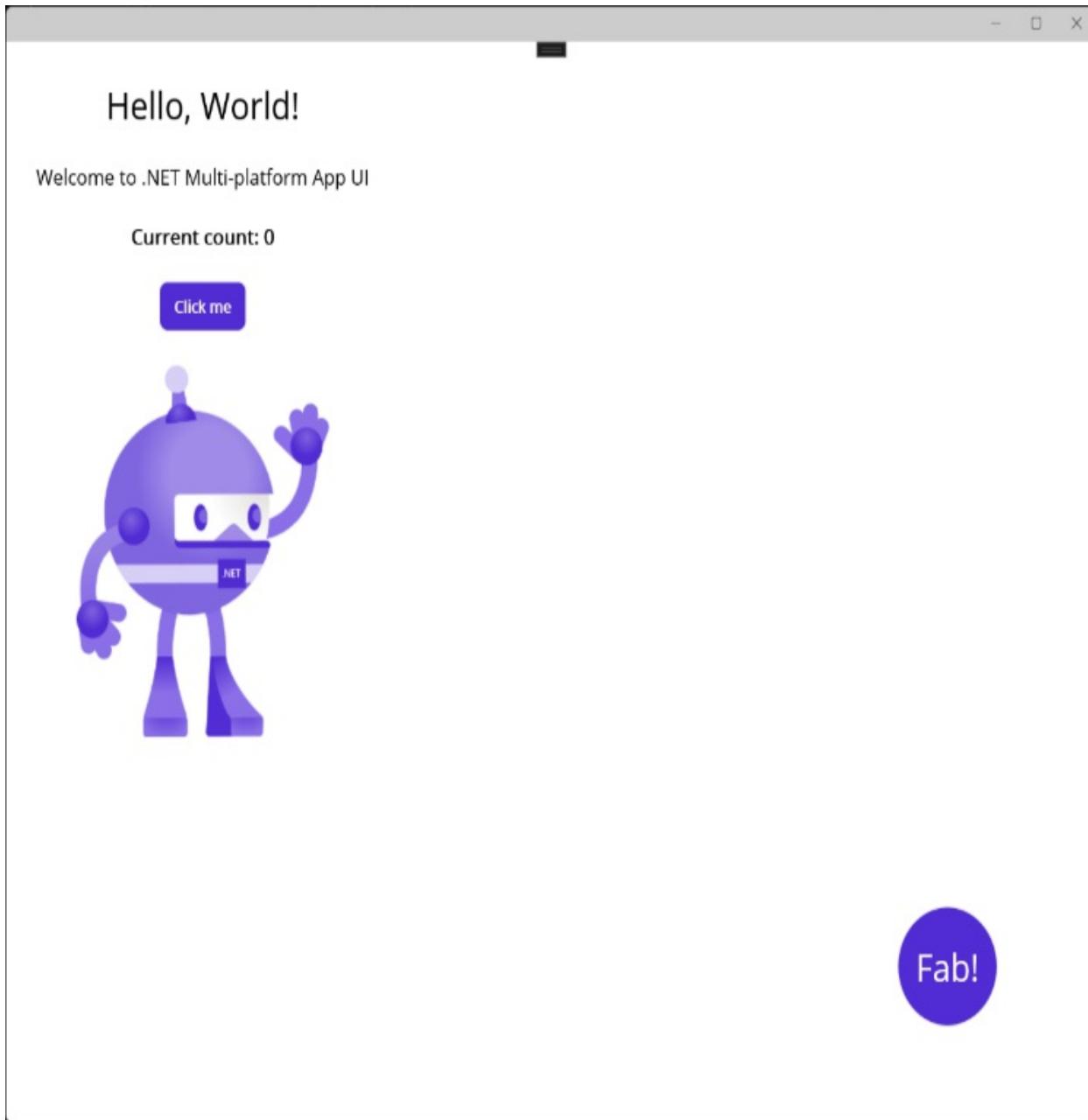
        </AbsoluteLayout>
    </ContentPage>

```

Run the app now and you should see something like figure 6.14

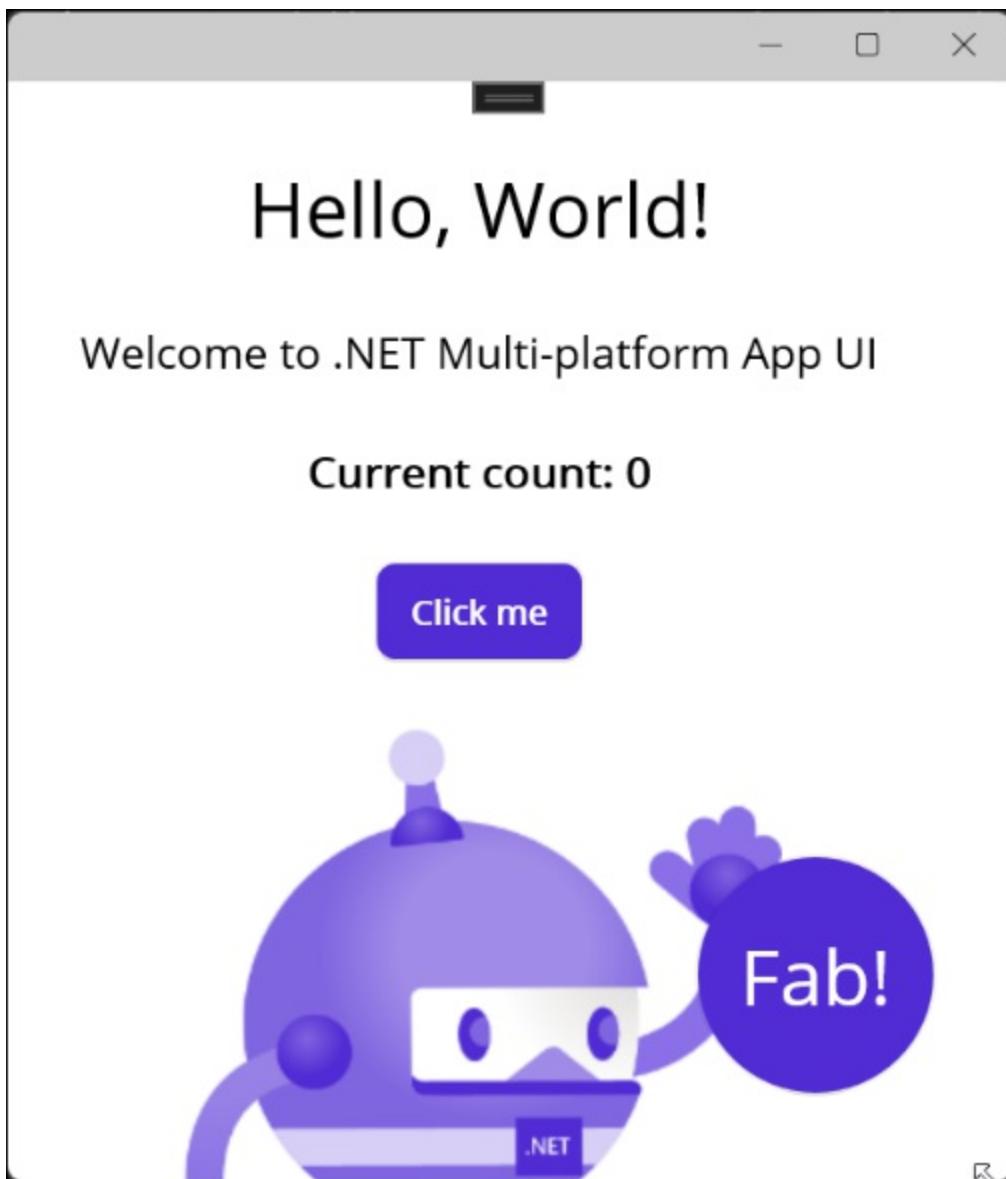
Figure 6.14 The contents of MainPage are wrapped in an AbsoluteLayout, which lets us position a FAB at 90% of the way down and across the screen. But now the rest of the contents are in the

top left.



You can resize the window as much as you like, and the FAB will always stay in the same relative position.

Figure 6.15 Resizing the window moves the FAB in relation to the other content but keeps its position relative to the AbsoluteLayout (which resizes with the window).



You'll notice that since we wrapped the `ScrollView` in an `AbsoluteLayout`, it's now positioned in the top left of the window. This is because, as mentioned above, it's now within an `AbsoluteLayout` and doesn't have any `LayoutBounds` specified. Therefore, it assumes the default values of 0 for the x and y positions at the top left, and automatic size.

There are other ways to create a FAB, and we'll see one in the next section. You'll probably find that you use `AbsoluteLayout` less frequently than other layouts, as most things that you can achieve with it are possible with other layouts. However, as a .NET MAUI developer it's a powerful tool to have in your belt.

You can find out much more about `AbsoluteLayout` in the .NET MAUI documentation.

6.4 Putting it all together

Throughout the remainder of this chapter, we're going to see how we can use the simple building blocks .NET MAUI provides by combining our familiar layouts and controls to create a rich UI. We'll put ourselves in the shoes of a UI engineer at Microsoft, having received the design for the Microsoft Outlook mobile app, and see how we can use the layouts we've learned, to arrange the controls we are no familiar with, to implement the design. Figure 6.16 shows the Microsoft Outlook app running on iOS.

Figure 6.16 Microsoft Outlook running on iOS.

08:40

Inbox

Focused Other Filter

 **Matt Goldman** 08:40
Message 8
Cras finibus pharetra vulputate. Praesent iaculis nulla ut pharetra ullamcorper. Suspendisse te...

 **Matt Goldman** 08:40
Message 7
Cras sit amet hendrerit massa. Donec aliquet, magna sed consectetur volutpat, metus mi da...

 **Matt Goldman** 08:40
Message 6
Maecenas in scelerisque sem. Vivamus id sem dictum, lacinia elit a, lobortis massa. Duis nec I...

 **Matt Goldman** 08:39
Message 5
Aliquam ante nibh, pellentesque vitae auctor vitae, porta sit amet est. Suspendisse ferment...

 **Matt Goldman** 08:39
Message 4
Praesent tincidunt mi lectus. Curabitur eleifend nulla vitae erat efficitur euismod. Vivamus id to...

 **Matt Goldman** 08:39
Message 3
Nullam purus risus, consequat eget placerat eget, accumsan vitae erat. Phasellus a ligula ul...

 **Matt Goldman**
Message 2
This message has no content.

 Email

 Search

 20 Calendar



We're not going to build any of the functionality Outlook provides, we're just going to recreate the UI of the Inbox screen to sharpen our layout skills in .NET MAUI.

UI Challenges

A UI challenge is where you take an existing UI design and build it in .NET MAUI (of course you could do the same with another UI framework). A common approach is to replicate the UI of a well-known, existing app (which we do in this section). Another is to find a fun and interesting concept design and build that; I often see Dribbble.com used as inspiration for these.

UI challenges are a popular way to keep your UI building skills sharp. They help you identify and solve UI problems and help you to build out the UI building tools you have in your mental toolkit. Several prominent content creators often post blogs or videos of their UI challenges, showing how they solve problems and create beautiful and functional designs. A great way to stay up to date with these is by subscribing to the Weekly Xamarin newsletter (<https://weeklyxamarin.com/>). Two of my favourites are Leomaris Reyes and Kym Phillipps. I recommend following them both for inspiration and to learn new UI tricks and skills and keep your eye out for others posting their UI challenges online. And, of course, do your own!

6.4.1 Adding the App's Shared Resources

Let's get started! Create a new .NET MAUI app from the `blankmaui` template and call it `OutlookClone`.

Outlook uses Microsoft's Fluent Design system, which includes an icon set. The icons are available in their own repository on GitHub, which you can find at <https://github.com/microsoft/fluentui-system-icons>, in various formats, but in `OutlookClone`, we're going to use fonts that contain all the symbols, and we'll see how we can use font icons.

I've already grabbed the two font assets we'll use in `OutlookClone`, and they are available in the book's online resources. Download the two font files, `FluentSystemIcons-Filled.ttf` and `FluentSystemIcons-Regular.ttf` and

import them into the Resources/Fonts folder. Next, we need to register the fonts in `MauiProgram.cs`. Listing 6.27 shows `MauiProgram.cs` for `OutlookClone`, with the added font registrations highlighted in **bold**.

Listing 6.27 MauiProgram.cs

```
namespace OutlookClone
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSa
                fonts.AddFont("OpenSans-Semibold.ttf", "OpenS
                fonts.AddFont("FluentSystemIcons-Filled.ttf",
                fonts.AddFont("FluentSystemIcons-Regular.ttf"
            });

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

These fonts are now registered as application-wide resources, accessible via the names registered for them in `MauiProgram.cs`. We'll see how to use these icons shortly, but we also need some other application-wide resources: colours. We could specify the colours individually for every control, but this is laborious and unnecessary.

I used a colour picker to identify the colours we need. We'll register these in the app's `ResourceDictionary`. We'll look into this in more detail in chapter 11, but for now, add the code from listing 6.28 in **bold** to `App.xaml`.

Listing 6.28 App.xaml

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

```

    xmlns:local="clr-namespace:OutlookClone"
    x:Class="OutlookClone.App">
<Application.Resources>
    <ResourceDictionary Source="Resources/Styles.xaml">
        <Color x:Key="Primary">#0878d3</Color>
        <Color x:Key="Secondary">#f1edec</Color>
        <Color x:Key="Tertiary">#717171</Color>
    </ResourceDictionary>
</Application.Resources>
</Application>

```

Each of these colours is now available to use as a `StaticResource` anywhere in the app. Now let's start breaking down the UI.

6.4.2 Defining the UI as a Grid

The first step is to break down the screen into its component parts and figure out how to replicate those parts with .NET MAUI. And the first of these is the screen itself. Figure 6.17 shows how we can use a `Grid` to break down the top-level components of the Outlook UI, consisting of four rows..

Figure 6.17 The Outlook Inbox UI broken down as rows in a Grid. We've ignored the status bar and safe area. Using this approach, we can see that we have four rows. The top row has the title and search, the next row which has the focused inbox switch and the filter button. At the bottom we can see the tab bar, and between the second and fourth rows is the list of messages, which takes up all the remaining space.



Next, we have the list of messages, and at the bottom we have the tab bar. In a real-world app, we would use Shell or a TabbedPage to provide these tabs

(which we'll look at in chapter 7) or, in other cases an external control library that includes tabs. In this case, as we're just replicating the UI and not using these for navigation, we'll build the tabs directly into the page.

Looking at this top-level Grid, I can see that the top row (row 0) is slightly larger than the second row (row 1), and that the last row (row 3) is slightly larger still. The middle row (row 2) takes up all the remaining space. I've measured these and a close enough approximation gives us row heights of 50, 40, * and 80. This is enough to get started, so let's add the top-level grid to `MainPage.xaml`. Listing 6.29 shows this initial scaffolding.

Listing 6.29 MainPage.xaml

```
<?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="OutlookClone.MainPage">

    <Grid RowDefinitions="50, 40, *, 80"> #A
        </Grid>
</ContentPage>
```

6.4.3 Creating the Title Bar with FlexLayout

We'll use a `FlexLayout` for the top row of the Grid, which will let us easily position the child items at the start and end using `SpaceBetween`. Inside this `FlexLayout` we can use `HorizontalStackLayout` to position the home icon and title at the start, and a `Label` to position the search icon at the end.

Figure 6.18 We can use a `FlexLayout` to arrange the top row, using `SpaceBetween` to position the child views on either side. On the left we can use a `HorizontalStackLayout` with two `Labels` (one for the icon and one for the title), and a `Label` for the search icon on the right.



As we're going to start using the icons now, we need a simple way to reference the icons we want to use. Usually when using a font, you just type

out the text that you want and let the font display it for you, but we're using glyphs rather than ASCII or Unicode characters. There are a few tools you can use to find the glyph codes you need, but I've already identified the glyphs we need (I used <https://andreinitescu.github.io/IconFont2Code/>). The table below shows the which glyphs we'll use from which font asset, and for what purpose.

Icon	Font	Glyph code
Home	FluentFilled	fa38
Search	FluentRegular	fb26
Filter	FluentRegular	f408
Mail	FluentFilled	f513
Calendar	FluentRegular	03de

Using these in XAML is easy. We set the `FontFamily` property of a `Label` to the font we want to use and set the `Text` property to the code of the glyph. When using these codes in XAML, we have to prefix them with the characters ‘&#x’ and end with ‘;’ (without the quotes).

Listing 6.30 shows the updated `MainPage.xaml`, with the inner layout for the top row added. The added lines are shown in **bold**.

Listing 6.30 MainPage.xaml with the top row layout added

```
<?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="OutlookClone.MainPage">

<Grid RowDefinitions="50, 40, *, 80">

    <FlexLayout Grid.Row="0"
                HorizontalOptions="FillAndExpand"
                VerticalOptions="FillAndExpand"
                BackgroundColor="{StaticResource Primary}"
                JustifyContent="SpaceBetween">

        <HorizontalStackLayout Margin="5, 0, 0, 0"
                               Spacing="10">

            <Label Text="༸""
                  FontFamily="FluentFilled"
                  TextColor="{StaticResource Primary}"
                  HorizontalTextAlignment="Center"
                  HorizontalOptions="StartAndExpand"
                  VerticalTextAlignment="Center"
                  BackgroundColor="White"
                  VerticalOptions="Center"
                  WidthRequest="30"
                  HeightRequest="30"
                  FontSize="Large"/>

            <Label Text="Inbox"
                  VerticalTextAlignment="Center"
                  HorizontalOptions="StartAndExpand"
                  TextColor="White"
                  FontAttributes="Bold"
                  FontSize="Large"/>

        </HorizontalStackLayout>

        <Label Text="ﬦ""
                  FontFamily="FluentRegular"
                  TextColor="White"
                  VerticalOptions="Center"
                  HorizontalOptions="EndAndExpand"
                  HorizontalTextAlignment="End"
                  WidthRequest="40"
                  FontSize="Large"
                  Margin="0, 0, 5, 0"/>

    </FlexLayout>

</Grid>
```

```
</ContentPage>
```

This gives us nearly everything we need for the top row, but if we run it now the home icon would be square, and it's round in Outlook; but we can easily clip the Label to make it round. Listing 6.31 shows the updated home icon Label, with ellipse geometry added to make the icon round. The changes are in **bold**.

Listing 6.31 The updated home icon

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

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            <HorizontalStackLayout ...>
                <Label Text="༸"**
                    FontFamily="FluentFilled"
                    TextColor="{StaticResource Primary}"
                    HorizontalTextAlignment="Center"
                    HorizontalOptions="StartAndExpand"
                    VerticalTextAlignment="Center"
                    BackgroundColor="White"
                    VerticalOptions="Center"
                    WidthRequest="30"
                    HeightRequest="30"
                    FontSize="Large">
                    <Label.Clip>
                        <EllipseGeometry RadiusX="15"
                            RadiusY="15"
                            Center="15, 15"/>
                    </Label.Clip>
                </Label>
                <Label .../>
            </HorizontalStackLayout>
            <Label .../>
        </FlexLayout>
    </Grid>
</ContentPage>
```

That completes the first row of the Grid. You can run OutlookClone now if you like; you will see a blank screen with just the top row. But we're ready to

move on to the second row.

6.4.4 Creating the Filter Bar with FlexLayout

Figure 6.19 We can use a FlexLayout to arrange the second row, using SpaceBetween to position the child views on either side. On the left is the focused inbox switch control, and on the right is a HorizontalStackLayout with two child Labels: one for the icon and the second for the word 'Filter'.



We'll use a FlexLayout for the second too. The focused inbox switch is a custom control and building that is outside the scope of this exercise, so we're going to use a little artistic license and replace it with a standard Switch and a Label. We'll place these inside a HorizontalStackLayout and use a second HorizontalStackLayout at the end of the FlexLayout for the filter icon and label.

Listing 6.32 shows the updated `MainPage.xaml` with the code for the second row added in **bold**. The code for the first row has been omitted for brevity.

Listing 6.32 The second row of the Grid

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

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            ...
        </FlexLayout>

        <FlexLayout Grid.Row="1"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="FillAndExpand"
            BackgroundColor="{StaticResource Primary}"
            Padding="20,5"
            JustifyContent="SpaceBetween">
```

```

<HorizontalStackLayout Margin="5,0,0,0">
    <Label Text="Focused"
        TextColor="White"
        VerticalOptions="Center"/>
    <Switch/>
</HorizontalStackLayout>

<HorizontalStackLayout Margin="0,0,5,0"
    Spacing="10">
    <Label Text=""
        FontFamily="FluentRegular"
        TextColor="White"
        VerticalOptions="Center"
        HorizontalOptions="EndAndExpand"
        HorizontalTextAlignment="End"
        WidthRequest="40"
        FontSize="Large"/>
    <Label Text="Filter"
        TextColor="White"
        VerticalOptions="Center"
        HorizontalOptions="EndAndExpand"
        HorizontalTextAlignment="End"/>
</HorizontalStackLayout>
</FlexLayout>

</Grid>
</ContentPage>

```

The header section of OutlookClone is now complete. If you run the app, you'll see a blank screen, with the header section, shown in figure 6.20, at the top.

Figure 6.20 The OutlookClone header section. Two rows are arranged using a VerticalStackLayout, and the rows themselves use FlexLayout to position items at the start and end with SpaceBetween.



6.4.5 Using Grid to create a FAB

The main section of the page is the message previews on row 2. Each message preview follows a specific layout and uses a data source to for each item. The collection of messages is scrollable, and a message can be selected. CollectionView is an ideal candidate for this use case.

Figure 6.21 Using CollectionView for the third row is a no-brainer, but we can see there's a floating action button in the bottom right-hand corner too. This can be in the same row (there's only one column) and we can use HorizontalOptions and VerticalOptions to position it at the end.



Matt Goldman

08:40

Message 8

Cras finibus pharetra vulputate. Praesent iaculis nulla ut pharetra ullamcorper. Suspendisse te...



Matt Goldman

08:40

Message 7

Cras sit amet hendrerit massa. Donec aliquet, magna sed consectetur volutpat, metus mi da...



Matt Goldman

08:40

Message 6

Maecenas in scelerisque sem. Vivamus id sem dictum, lacinia elit a, lobortis massa. Duis nec l...



Matt Goldman

08:39

Message 5

Aliquam ante nibh, pellentesque vitae auctor vitae, porta sit amet est. Suspendisse ferment...



Matt Goldman

08:39

Message 4

Praesent tincidunt mi lectus. Curabitur eleifend nulla vitae erat efficitur euismod. Vivamus id to...



Matt Goldman

08:39

Message 3

Nullam purus risus, consequat eget placerat eget, accumsan vitae erat. Phasellus a ligula ul...



Matt Goldman

Message 2



We're not going to build the `CollectionView` just yet as we'll need some data to see any contents, which we'll get to shortly. But we can add the floating action button (FAB). We'll add it to row 2 and position it vertically and horizontally at the end. As we're using a `Button`, which has a `CornerRadius` property, we won't need to do any clipping to make it round. And we'll give it a `Shadow` to make it mimic the real app.

Listing 6.33 shows the updated `MainPage.xaml` with the added `Button` in **bold**. Most of the rest of the code has been omitted for brevity.

Listing 6.33 The FAB

```
<?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="OutlookClone.MainPage">

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            ...
        </FlexLayout>

        <FlexLayout ...>
            ...
        </FlexLayout>

        <Button Grid.Row="2"
            BackgroundColor="{StaticResource Primary}"
            HorizontalOptions="EndAndExpand"
            VerticalOptions="EndAndExpand"
            Margin="20"
            HeightRequest="60"
            WidthRequest="60"
            CornerRadius="30"
            FontSize="30"
            Text="+"
            >
            <Button.Shadow>
                <Shadow Brush="Black"
                    Offset="5,5"
                    Radius="10"
                    Opacity="0.5"/>
            </Button.Shadow>
        </Button>
    </Grid>
```

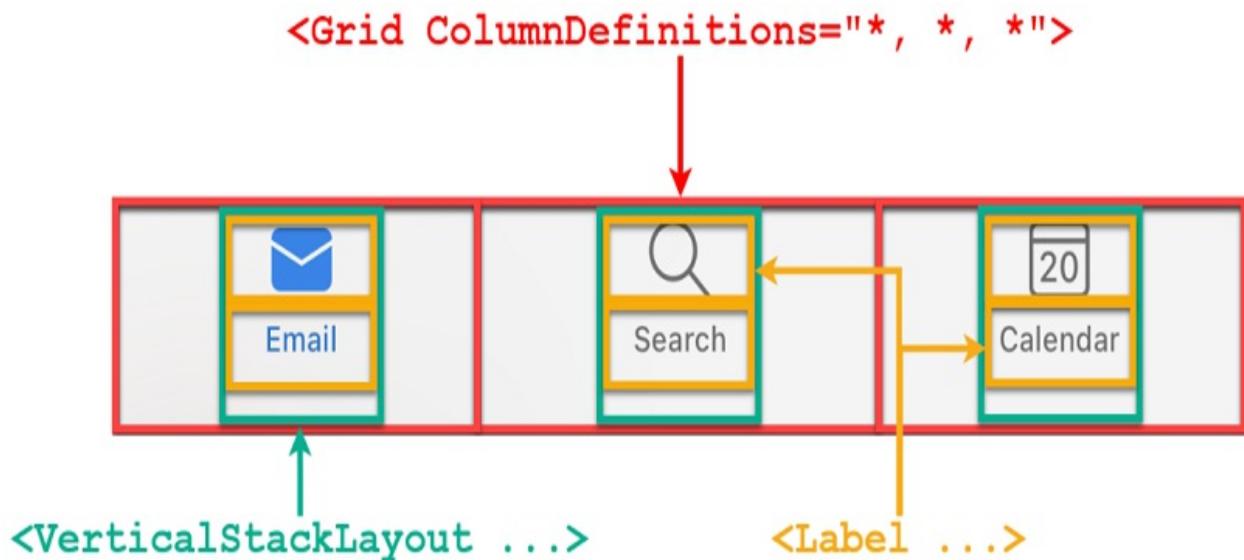
```
</ContentPage>
```

Now that we've got the FAB, the final component before we start populating the messages section is the tab bar footer.

6.4.6 Building a Tab Bar with Grid

To build the tab bar, we'll add a `Grid` to the fourth row (row 3) of the page's top-level `Grid`. It will have three columns, one for each tab, and within each column will be a `VerticalStackLayout` to arrange the tab's icon and label.

Figure 6.22 The last row is the tab bar. We'll use another `Grid` for this part, with three columns. In each column we'll use a `VerticalStackLayout` with two `Labels` as child items; one for the tab icon and one for the tab label.



Let's start by adding the `Grid`. Listing 6.34 shows the code for `MainPage.xaml` with the tab bar `Grid` added in **bold**. Most of the remaining code has been omitted for brevity.

Listing 6.34 The Tab bar

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

```

<Grid RowDefinitions="50, 40, *, 80">
    <FlexLayout ...>
        ...
    </FlexLayout>

    <FlexLayout ...
        ...
    </FlexLayout>

    <Button ...>
        ...
    </Button>

    <Grid Grid.Row="3"
        HorizontalOptions="FillAndExpand"
        VerticalOptions="FillAndExpand"
        ColumnDefinitions="*, *, *"
        Padding="5"
        BackgroundColor="{StaticResource Secondary}">

        ...
    </Grid>
</ContentPage>

```

That defines the high-level layout for the tab bar. Now let's add the `VerticalStackLayout` for the first tab. Listing 6.35 shows the updated `MainPage.xaml`, with the added code in **bold**. Some of the code (including the `Grid`'s properties) has been removed for brevity.

Listing 6.35 The first tab

```

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

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            ...
        </FlexLayout>

        <FlexLayout ...
            ...
        </FlexLayout>

```

```

<Button ...>
    ...
</Button>

<Grid ...>
    <VerticalStackLayout HorizontalOptions="Center"
    Grid.Column="0">

        <Label Text="" 
            FontFamily="FluentFilled"
            TextColor="{StaticResource Primary}"
            HorizontalTextAlignment="Center"
            HorizontalOptions="Center"
            VerticalTextAlignment="Center"
            VerticalOptions="Center"
            WidthRequest="30"
            HeightRequest="30"
            FontSize="30"/>

        <Label Text="Email"
            TextColor="{StaticResource Primary}"
            HorizontalTextAlignment="Center"
            HorizontalOptions="Center"
            VerticalTextAlignment="Center"
            VerticalOptions="Center"
            FontSize="11"/>
    </VerticalStackLayout>

</Grid>
</ContentPage>

```

Now that we've got the first tab complete, we can copy and paste the code for the tab to add the second and third tabs. We'll need to change the column, glyph and text colors.

Listing 6.36 shows the updated `MainPage.xaml` file with the second and third tabs added in **bold**. The first tab and most of the remaining code has been omitted for brevity. Pay attention to the columns, `Text` properties, and `TextColor` properties.

Listing 6.36 The final two tabs

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

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

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            ...
        </FlexLayout>

        <FlexLayout ...>
            ...
        </FlexLayout>

        <Button ...>
            ...
        </Button>

        <Grid ...>
            <VerticalStackLayout ...>
                ...
            </VerticalStackLayout>

            <VerticalStackLayout HorizontalOptions="Center"
                Grid.Column="1">

                <Label Text="#"&#xfb26;""
                    FontFamily="FluentRegular"
                    TextColor="{StaticResource Tertiary}"
                    HorizontalTextAlignment="Center"
                    HorizontalOptions="Center"
                    VerticalTextAlignment="Center"
                    VerticalOptions="Center"
                    WidthRequest="30"
                    HeightRequest="30"
                    FontSize="30"/>

                <Label Text="Search"
                    TextColor="{StaticResource Tertiary}"
                    HorizontalTextAlignment="Center"
                    HorizontalOptions="Center"
                    VerticalTextAlignment="Center"
                    VerticalOptions="Center"
                    FontSize="11"/>

            </VerticalStackLayout>

            <VerticalStackLayout HorizontalOptions="Center"
                Grid.Column="2">

                ...
            </VerticalStackLayout>
        </Grid>
    </Grid>
</ContentPage>
```

```

        Grid.Column="2">

    <Label Text="Ϟ" 
        FontFamily="FluentRegular"
        TextColor="{StaticResource Tertiary}"
        HorizontalTextAlignment="Center"
        HorizontalOptions="Center"
        VerticalTextAlignment="Center"
        VerticalOptions="Center"
        WidthRequest="30"
        HeightRequest="30"
        FontSize="30"/>

    <Label Text="Calendar"
        TextColor="{StaticResource Tertiary}"
        FontSize="11"
        HorizontalTextAlignment="Center"
        HorizontalOptions="Center"
        VerticalTextAlignment="Center"
        VerticalOptions="Center"/>

    </VerticalStackLayout>
</Grid>

</Grid>
</ContentPage>

```

That completes the layout for OutlookClone. We've still got to add the messages, but you can run it now and immediately see what we're building. If you run the app now you should see something like figure 6.23.

Figure 6.23 The OutlookClone with its layout complete, just missing the messages



The last piece of the puzzle is the `collectionView` that will occupy the main

section of the page. Before we build that though, let's set up some data to display messages.

6.4.7 Populating Dummy Data

We're not going to build an email client, so we need a way to simulate the data for our inbox. For this, we'll use an API that returns random quotes from *The Simpsons*. The API will give us a quote along with the name of the character who said it and a link to a picture of them. We'll use the name as the sender, the picture for the avatar, and the quote can be both the message and subject.

The API that we're going to use is <https://thesimpsonsquoteapi.glitch.me>. By checking the API documentation, we can see that we can call the quotes endpoint specifying count as a query parameter. An initial call to that endpoint gives us the structure of the JSON data that it returns, and from there we can create a class to use in our code to deserialise the response from the API.

Note

As mentioned in 6.2.1, there are a few ways you can generate the C# classes from the Simpsons quote API's JSON data, including the built in tooling in Visual Studio or free online converters.

Create a class in the OutlookClone project called Simpson. Listing 6.37 shows the code for this class.

Listing 6.37 The Simpson class

```
namespace OutlookClone;

public class Simpson
{
    public string quote { get; set; }
    public string character { get; set; }
    public string image { get; set; }
    public string characterDirection { get; set; }
}
```

Now that we've got this class, we can write some logic to call the API and build a collection of quotes. In the `MainPage.xaml.cs` file, add an `ObservableCollection` of type `Simpson` and add a field for the URI we use to call the API.

Next, override the `OnAppearing` method and make it `async`. In this method, we'll instantiate an `HttpClient` and use it to call the API, returning the data as a collection of the `Simpson` type we've defined. Then, we can iterate through the results and add each one to the `ObservableCollection`.

Listing 6.38 shows the updated `MainPage.xaml.cs` file, with the added code in **bold**.

Listing 6.38 MainPage.xaml.cs

```
using System.Collections.ObjectModel;
using System.Net.Http.Json;

namespace OutlookClone
{
    public partial class MainPage : ContentPage
    {

        private string contentUri = "https://thesimpsonsquoteapi.

        public ObservableCollection<Simpson> Simpsons = new();

        public MainPage()
        {
            InitializeComponent();
        }

        protected override async void OnAppearing()
        {

base.OnAppearing();

            var httpClient = new HttpClient();

            var jsonResponse = await httpClient.GetFromJsonAsync<

                jsonResponse.ForEach(s => Simpsons.Add(s));
            }
        }
    }
}
```

```
}
```

Now that we've set up our data source and know the structure of the data we're displaying, we can add the `CollectionView` and use that structure to define our `DataTemplate`.

6.4.8 Displaying the Messages with CollectionView

Now that we've got a data source, let's add the `CollectionView` to the UI. Open `MainPage.xaml` and add a `CollectionView` to row 2 of the page's top-level `Grid`.

You'll need to add it **before** the `Button` in the same row, otherwise the `CollectionView` will get rendered on top of the `Button` (you can also adjust the z-index of these elements, but for a simple UI like this it's easier to just add them in the right order).

Listing 6.39 shows `MainPage.xaml` with the `CollectionView` added. Some of the code has been omitted for brevity, and the added code is shown in **bold**.

Listing 6.39 `MainPage.xaml` with the `CollectionView` added

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

    <Grid RowDefinitions="50, 40, *, 80">
        <FlexLayout ...>
            ...
        </FlexLayout>

        <FlexLayout ...>
            ...
        </FlexLayout>

        <CollectionView Grid.Row="2"
            x:Name="MessageCollection"
            HorizontalOptions="Fill"
            VerticalOptions="Fill">
            <CollectionView.ItemTemplate>
                <DataTemplate>
```

```

        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

<Button ...>
    ...
</Button>

<Grid ...>
    ...
</Grid>

</Grid>
</ContentPage>

```

With the `CollectionView` in place, it's time to define the template for the items in the collection. Looking at the original Outlook UI, we can see that there is an avatar image, a sender name, a message subject, a preview of the message body, and a time or day when the message was sent.

We can use a `Grid` for this layout. The `Grid` will have three columns; the first and last column will have a width of 50, and the middle column can take up the remaining space. We'll have three rows, with the second row being slightly less high than the top row, and then the third row being twice the height of the second.

Figure 6.24 The message template for the Inbox can be laid out as a `Grid`, with three columns and three rows. The avatar will reside in the first column and row and will have a row span of 3 (there's nothing else in the first column). The sender's name will go into the first row, second column. The subject will go into the second row, second column, and the message body preview will go into the third row, second column, and will span into the third column too. The time or day when the message was sent will go into the first row, third column.

	Matt Goldman	08:39
	Message 5	
	Aliquam ante nibh, pellentesque vitae auctor vitae, porta sit amet est. Suspendisse ferment...	

Now that we know how to lay out the message template, let's add this to the `CollectionView`. We haven't set the binding yet, but we know that we're going to use our `Simpson` class as the source data type, so we can bind to properties on that class in the template. Listing 6.40 shows the code for the data template, with the added code in **bold**. The rest of the page's code has been omitted for brevity.

Listing 6.40 MainPage.xaml with the message template

```
<CollectionView ...>
    <CollectionView.ItemTemplate>
        <DataTemplate>
<Grid ColumnDefinitions="50, *, 50"
      RowDefinitions="25, 20, 40"
      HorizontalOptions="Fill"
      VerticalOptions="Fill"
      Padding="10, 5, 20, 5">

    <Image WidthRequest="40"
          HeightRequest="40"
          Grid.RowSpan="3"
          VerticalOptions="Start"
          HorizontalOptions="Start"
          Aspect="AspectFill"
          Source="{Binding image}">
      <Image.Clip>
        <EllipseGeometry RadiusX="20"
                         RadiusY="20"
                         Center="20, 20"/>
      </Image.Clip>
    </Image>

    <Label Grid.Row="0"
          Grid.Column="1"
          Text="{Binding character}"
          FontSize="18"
          FontAttributes="Bold"
          TextColor="Black"/>

    <Label Grid.Row="1"
          Grid.Column="1"
          Text="{Binding quote}"
          LineBreakMode="TailTruncation"
          VerticalOptions="Start"
          TextColor="Black"/>
```

```

        <Label Grid.Row="2"
               Grid.Column="1"
               Grid.ColumnSpan="2"
               Text="{Binding quote}"
               LineBreakMode="WordWrap"
               VerticalOptions="Start"
               TextColor="{StaticResource TertiaryColor}" />

        <Label Grid.Row="0"
               Grid.Column="2"
               Text="Saturday"
               FontSize="12"
               TextColor="{StaticResource TertiaryColor}" />
    </Grid>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>

```

This completes the layout for the message template. We need to wire up the `ItemsSource` property of the `CollectionView`, but before we do, I want to make one more addition for a slight UX improvement.

Add an `ActivityIndicator` to the page's top-level `Grid`. We'll use this to show that data is loading. We'll place it in row 2 so that it we can show it where the messages will be shown once the data is loaded. We'll control the visibility of the `ActivityIndicator` in the code behind.

Listing 6.41 shows `MainPage.xaml` with the `ActivityIndicator` added, shown in bold. The rest of the code has been omitted for brevity.

Listing 6.41 MainPage.xaml with the ActivityIndicator

```

<?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="OutlookClone.MainPage">

    <Grid RowDefinitions="50, 40, *, 80">
        ...
        <ActivityIndicator Grid.Row="2"
                           Color="{StaticResource Primary}" />
    </Grid>
</ContentPage>

```

```

        IsRunning="True"
        IsEnabled="True"
        VerticalOptions="Center"
        HorizontalOptions="Center"
        x:Name="LoadingIndicator"/>

    </Grid>
</ContentPage>
```

The `MainPage.xaml` UI is now 100% complete. The last steps are to set the `ItemsSource` property of the `CollectionView`, and show/hide the `ActivityIndicator` as appropriate. We'll do both of these in the code behind. Make the changes shown in **bold** in listing 6.42 to `MainPage.xaml.cs`.

Listing 6.42 The complete MainPage.xaml.cs

```

using System.Collections.ObjectModel;
using System.Net.Http.Json;

namespace OutlookClone
{
    public partial class MainPage : ContentPage
    {

        private string contentUri = "https://thesimpsonsquoteapi.

        public ObservableCollection<Simpson> Simpsons = new();

        public MainPage()
        {
            InitializeComponent();
            MessageCollection.ItemsSource = Simpsons;
        }

        protected override async void OnAppearing()
        {
            LoadingIndicator.IsVisible = true;

            base.OnAppearing();

            var httpClient = new HttpClient();

            var jsonResponse = await httpClient.GetFromJsonAsync<
```

```
        jsonResponse.ForEach(s => Simpsons.Add(s));  
        LoadingIndicator.IsVisible = false;  
    }  
}  
}
```

The OutlookClone app is now complete. Go ahead and run it, and you should see something like figure 6.25

Figure 6.25 The completed OutlookClone app



Replicating app UIs is a fun and challenging way to keep your UI skills sharp with .NET MAUI. There are also a bunch more things we could do with this app – add `SwipeView` to the messages, make the tabs a templated control (we'll see how to do this in chapter 8) or even, if you're feeling ambitious,

replicate the focused inbox switch from the original app.

6.5 Summary

- `Grid` is a versatile layout you can use for nearly any UI. You can build complex layouts using a `Grid`, like SSW.Rewards or even Microsoft Outlook, not just simple rows and columns like MauiCalc.
- With `BindableLayout`, you can use a data source to add child items to any layout using a `DataTemplate`, just like with `CollectionView`.
- `FlexLayout` is a great layout to use with `BindableLayout`. `HorizontalStackLayout` and `VerticalStackLayout` are good candidates for `BindableLayout` too, although `Grid` is not a sensible choice for this.
- With `AbsoluteLayout` you can gain precise control over where things are positioned on screen and how they are sized. Within an `AbsoluteLayout`, you can use proportional or absolute sizing and positioning to arrange views.

7 Pages and navigation

This chapter covers:

- How to break your app up into pages using the `ContentPage` base class
- How to use the navigation paradigms supported by .NET MAUI to navigate between different pages in your app
- Using Shell to simplify organising the pages in your app
- Passing data between pages when navigating

So far, we've built a handful of apps, all using a single page. This has worked well for the sample apps we've been building and works equally well for several commercial apps in the real world. But often, you need more than one page, for example to distinguish different areas of an app or logically group functionality. This becomes especially necessary with non-trivial apps when they grow too much to cram everything into a single screen.

In this chapter we'll look at navigation paradigms and the different ways that .NET MAUI supports providing multiple pages in your apps and navigating between them.

7.1 ContentPage

`ContentPage` is the most important Page type, in that the other Page types are just containers that provide different navigation paradigms that offer a way to present a `ContentPage`.

`ContentPage` has a single public property of type `View` called `Content`, which is what the page will render on screen. In XAML, the `Content` property is assigned by placing the XAML element you want to assign to this property between the opening and closing `<ContentPage...>...</ContentPage>` tags, as in listing 7.1. In C#, you can assign it just like any other property, as in listing 7.2.

Listing 7.1 ContentPage.xaml

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui  
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam  
    x:Class="MayApp.MyPage">  
  
<!--Your page's content goes here -->#A  
  
</ContentPage>
```

Note that you can also explicitly add the `Content` property and then add your views to it, by adding the `<ContentPage.Content>...` `</ContentPage.Content>` tags. However, this does not need to be explicitly specified. A view added between the `<ContentPage>` tags will be assigned to the `Content` property.

Listing 7.2 ContentPage.xaml.cs

```
namespace MyApp;  
  
public partial class MyPage : ContentPage  
{  
    public MyPage()  
{  
    InitializeComponent();  
    Content = new VerticalStackLayout();#A  
    }  
}
```

When building a Page in a .NET MAUI app, you will choose one of these two approaches (listings 7.1 and 7.2 would not be from the same app).

UI in C#

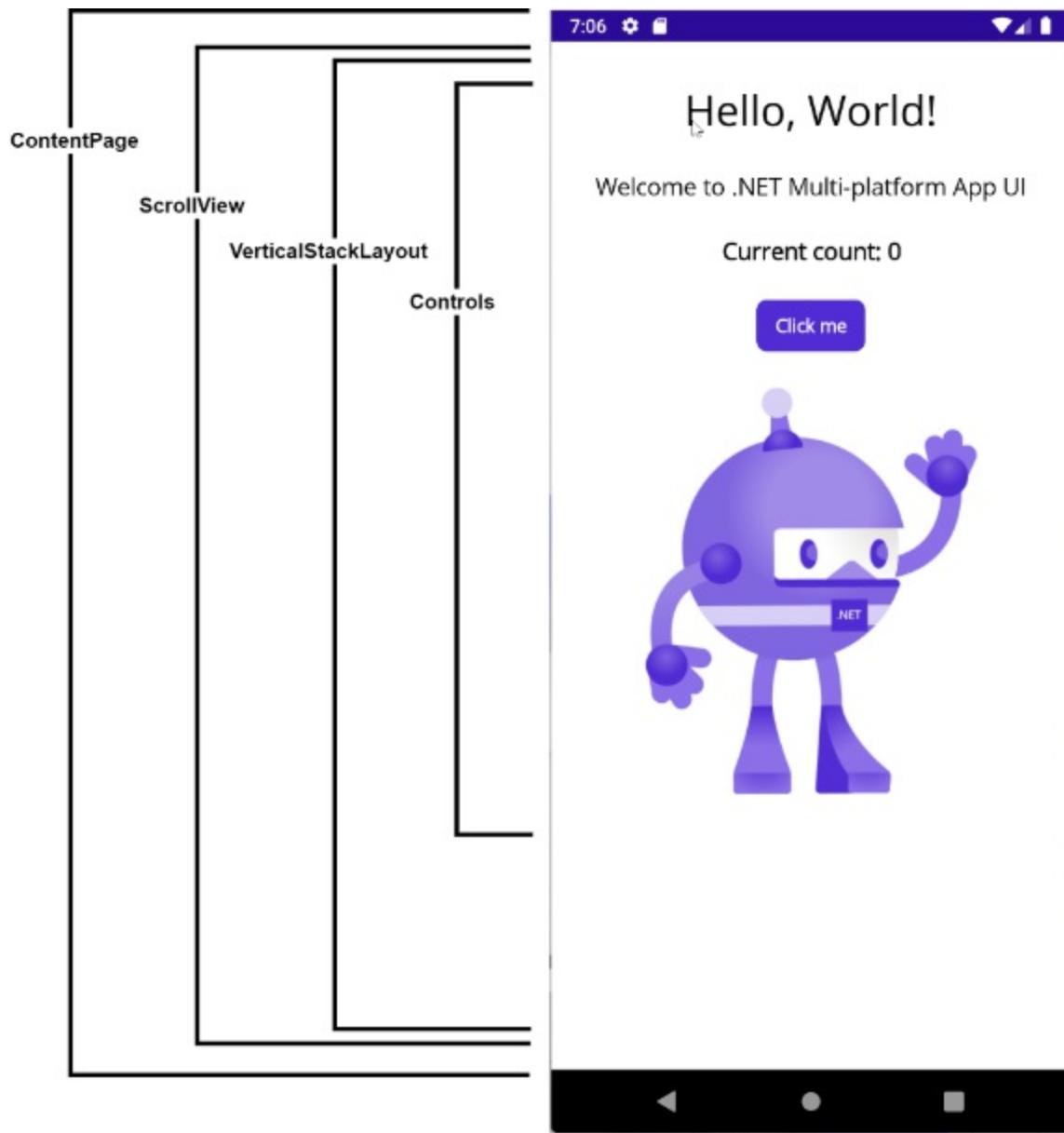
In this book, while we do dip into C# to declare or manipulate UI when necessary, we'll be focusing primarily on the XAML approach, but if you are interested in declaring your UI in C# code, you can find out more about it in the .NET MAUI documentation.

There are also plenty of community generated resources for learning and using C# declared UI, including markup extensions in the Community Toolkit that let you write your UI in C# using a fluent API. If this approach interests you, I recommend starting with the video by Gerald Versluis,

available on his YouTube channel at <https://www.youtube.com/watch?v=nCNh9G-Q688>.

All the Layouts and Controls provided in .NET MAUI are derived from the `View` base class, so can be assigned to the `View` property of `ContentPage`. It's common to assign a layout to the `Content` property of a `ContentPage`, or a `ScrollView` which we discussed in chapter 4. This is the approach we've seen in every example we've looked at so far, including the page generated by the template, as you can see in figure 7.1.

Figure 7.1 A ContentPage has a Content property to which you can assign a View, which means any layout or control. In this case, a ScrollView is assigned to the Content property, which wraps a VerticalStackLayout, which contains all the controls. The ScrollView ensures any content that doesn't fit on-screen is still available, and the VerticalStackLayout enables multiple controls to be added to the page.



As I mentioned, all controls in .NET MAUI are also derived from `View`, so you could in fact just add, for example, a `Button` to the `View` property of a `ContentPage`, as in listing 7.3.

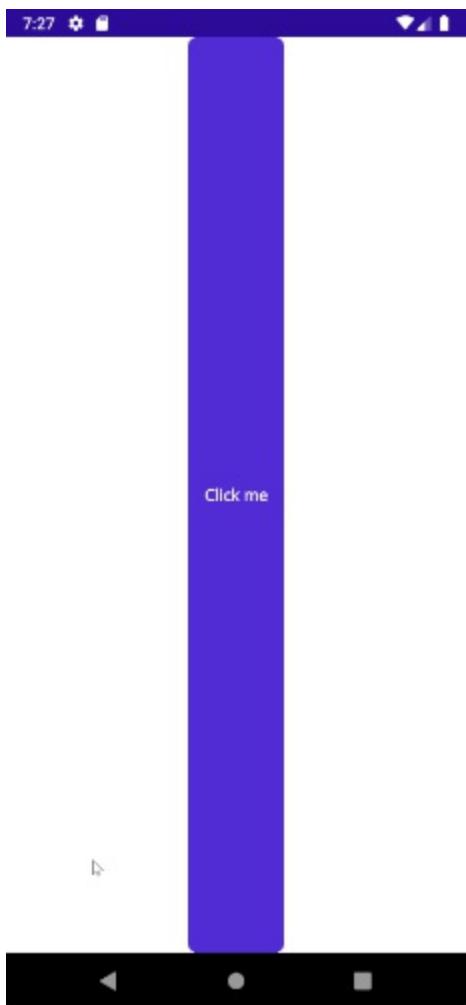
Listing 7.3 A Page with a single control as its Content

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

```
<Button Text="Click me"  
       FontAttributes="Bold"  
       HorizontalOptions="Center" />  
</ContentPage>
```

As you can see from figure 7.2, this wouldn't make for a particularly interesting page.

Figure 7.2 A ContentPage with a single View, in this case a Button, assigned to its Content property by placing the View between the ContentPage's opening and closing tags



While this is a valid approach to building .NET MAUI apps, it's not a sensible one, and you should stick to assigning either a layout or a ScrollView to the Content property of your pages.

7.2 Common Page Properties

In .NET MAUI, a `Page` is a class that inherits the `Page` base class (or in the case of `TabPage`, a collection of `Pages`). The `Page` base class has several properties related to rendering a UI on screen, as well as some lifecycle methods and events.

The `Page` base class has a few properties which are inherited by, and can be useful in, the derived page types. Depending on the navigation paradigm you're using to present your page, these properties may do different things, and in some cases nothing at all.

These properties are all bindable, meaning you can assign values to them directly or bind them to properties of corresponding types in a binding context. This section lists out the most useful and most commonly used of these properties.

Note

The `IsBusy` property isn't covered here because it does not work reliably and should not be used. Adding your own `ActivityIndicator` is trivial; you should do this instead, as we've seen in `MauiTodo` and `MauiMovies`.

7.2.1 IconImageSource

`IconImageSource` lets you specify an image to be displayed as the page's icon. If your page is being presented by either a `FlyoutPage` or `TabPage`, the image that you specify here will be displayed as the page's icon either in the Flyout menu or the TabBar.

7.2.2 Background Images

The `BackgroundImageSource` property of a page lets you specify an image that will be displayed in the background. Layouts, by default, are transparent, so any areas of the page not covered by a control will show the image specified as the background.

Image properties can be provided to specify how the image should be displayed (e.g., filled, tiled, etc). If you want a tiled background image, this can be a good option. However, setting a `BackgroundImageSource` can be yield unpredictable results if you want to fit or fill the image.

If you want a filled or fitted background image, you'll get much more reliable results using a `Grid` as a parent layout for your page and placing an `Image` in row and column 0 and setting the `Aspect` property that you want. If you don't want to use a `Grid` as your layout, you can still use this approach; simply add the `Image` and your preferred layout, in that order, to your page. You don't even need to define any rows or columns or specify which row and column the `Image` and your layout should go in, as by default the `Grid` will have a single row and column, and views inside a `Grid` will by default be in row and column 0.

Let's use this approach to add a background image to `MauiMovies`. I'm going to use a picture of the Hollywood sign that I got from [Pexels.com](#), a website for free stock images and other creative works.

Listing 7.4 MainPage.xaml with background image

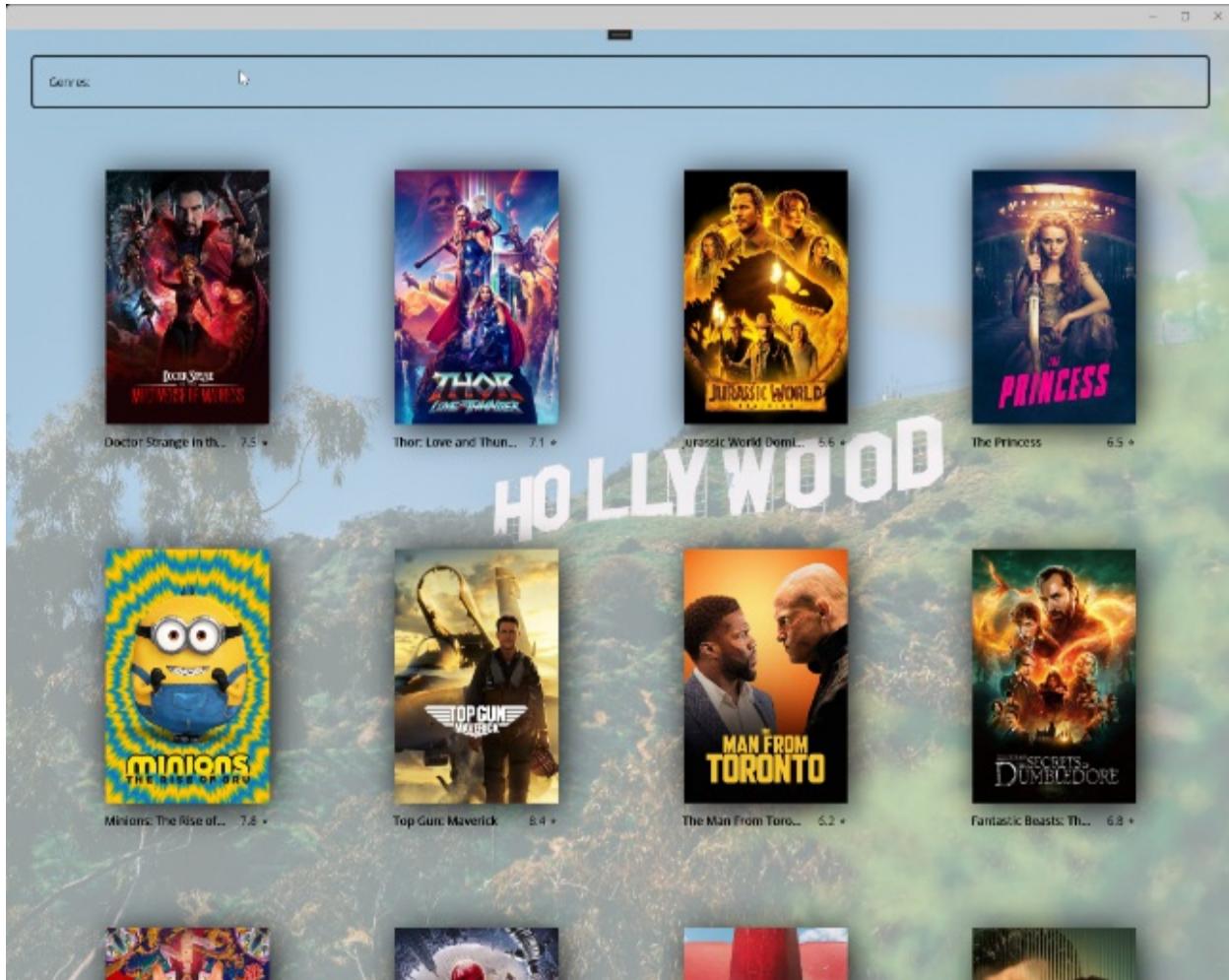
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Name="MoviePage"
              x:Class="MauiMovies.MainPage">

    <Grid>
        <Image Source="hollywood.png"
               Aspect="AspectFill"
               Opacity="0.4"/>

        <VerticalStackLayout Spacing="10" Padding="30">
            ...
        </VerticalStackLayout>
    </Grid>
</ContentPage>
```

Figure 7.3 A background image added to a page by assigning both the image and the page's layout to a single row and column Grid that takes up the whole page. This approach lets you reliably fit or fill the image, which yields unpredictable results when using the page's `BackgroundImageSource` property. Tiling a background image is more reliable when using

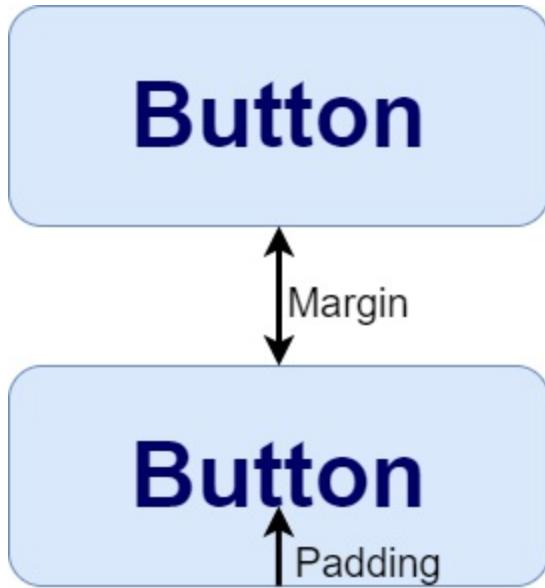
BackgroundImageSource.



7.2.3 Padding

Page has a Padding property of type Thickness. All views have a boundary, and Padding specifies how close to that boundary items inside the view are allowed to get.

Figure 7.4 Two buttons on a page. The arrow between them represents margin and specifies how apart views are from other views. The arrow pointing inward represents padding, and specifies how far from a view's boundary its internal elements can be displayed.



Padding differs from Margin in that Padding represents the space inside a view's boundary, whereas Margin corresponds to the space outside a view's boundary.

Specifying the page's Padding property will ensure a gap between the outer boundary of the page and any child items your page displays.

7.2.4 Title

Page has a `Title` property of type `string`. The Page's `Title` property is used to show which page of the app is currently being displayed, and is shown in the app's toolbar or navigation bar.

Figure 7.5 The MauiTodo app with the `Title` property of the `MainPage` set in XAML, with the title shown in the Navigation bar.

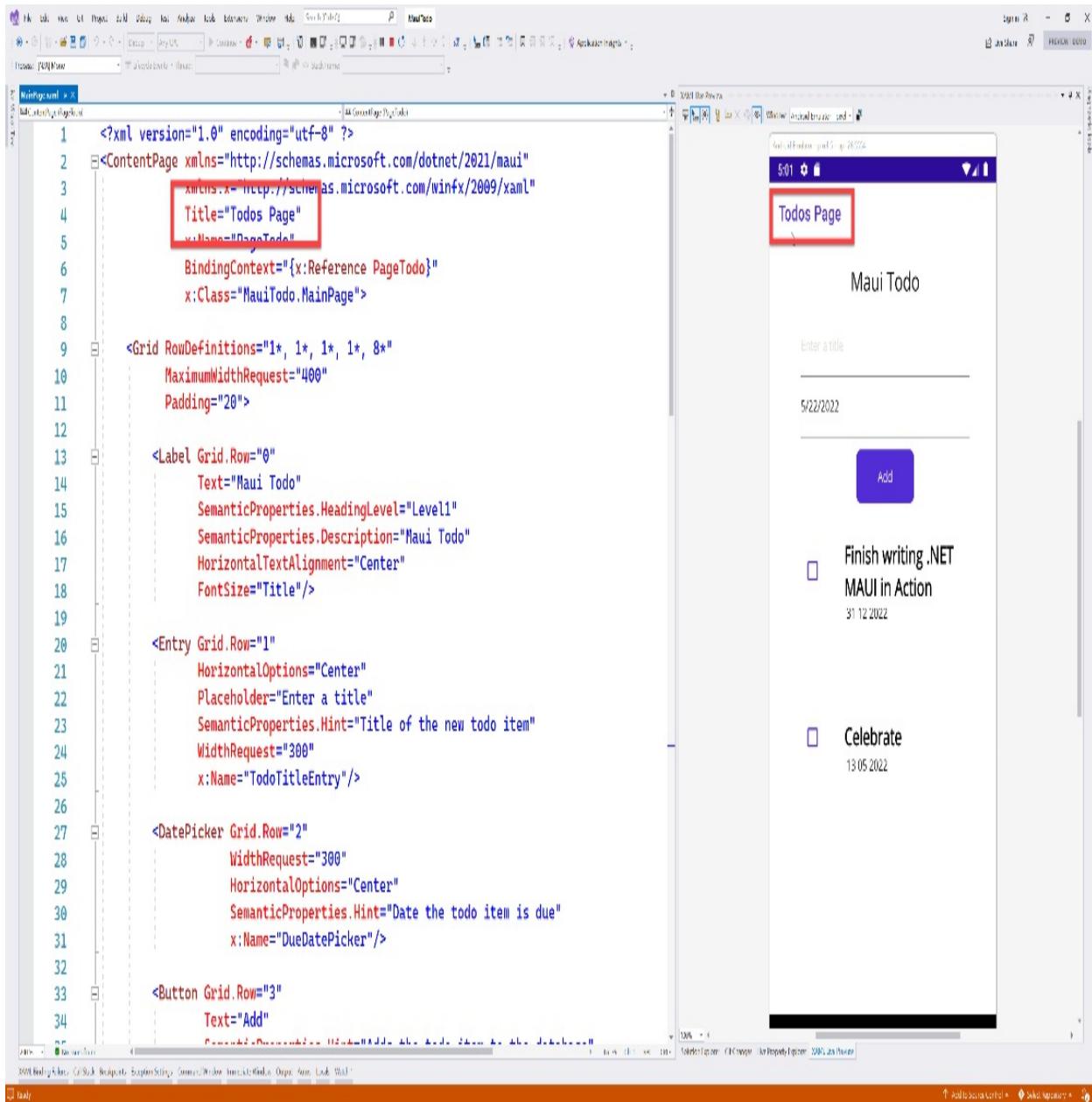


Figure 7.5 shows that the MainPage of the MauiTodo app would look like with a title set.

Note

In the example shown in figure 7.15, the MainPage has been enclosed in a NavigationPage. If you only make the change shown in the XAML here, you won't see the same result. We'll see how to do this later in the chapter.

With a single page app, as the title of the app and the title of the page are likely the same thing, displaying the page `Title` may not be necessary. In fact, due to the limitations of showing this property in the navigation bar, it's probably preferable to use a `Label` (or graphic of some kind) to show the title of your app or page in the page's content, as we've done in FindMe and MauiTodo.

For a multiple page app, displaying the page's title in the navigation bar is a quick and familiar way for your user to see where in the app they are. If you are using one of the navigation paradigms covered in this chapter, setting the `Title` property of the page will automatically make that title display in the navigation bar.

7.2.5 `MenuBarItems`

`MenuBarItems` is a collection of type `MenuBarItem` and is used on the desktop target platforms (Windows and macOS) to display a menu. The menu follows the standard presentation paradigm for each OS, so on Windows this will appear at the top of the window, and on macOS appears at the top of the screen. The `MenuBarItems` collection is not rendered on the mobile target platforms (iOS and Android).

A menu is a well-known interaction method for accessing features of an application, dating back to the earliest graphical user interfaces (GUIs). In a mobile app, UIs need to be simplified, to cater both for smaller screens and for fingers, which are more coarse-grained input devices than a mouse and pointer. But, in a desktop application, A menu is a good choice, especially in line-of-business (LOB) applications, or any application that offers a rich range of features which would be difficult to cram all onto one screen as buttons.

We'll see menus in action in chapter 10 when we look at catering your app for use on desktop operating systems.

7.3 Common Page Lifecycle Methods

The `Page` base class provides several methods that are called by event

handlers in response to page lifecycle events. Lifecycle events are events that are attached to the lifecycle of the page, as opposed to in direct response to an external action like an interaction from a user, or a push notification. Many of these lifecycle events do ultimately originate from a user interaction or some external stimulus, for example the `OnAppearing` method is called when a page appears, but a page only appears because a user has navigated to it. However, the method is attached to the lifecycle event of the page, not to the user interaction.

This section covers the lifecycle methods you are most likely to use in your apps. The .NET MAUI documentation has full coverage of all lifecycle methods; you can read more about them here: <https://docs.microsoft.com/en-us/dotnet/maui/fundamentals/app-lifecycle>.

7.3.1 OnAppearing

The `OnAppearing` method is called, as the name suggests, when a page appears. This is an important method as it is where you should perform any of your page initialisation logic. Like any class, `Page` and its derived types have constructors, and it can be tempting to perform initialisation logic here. But this is not what constructors are for; in a .NET MAUI `Page`, constructors should only be used for setting initial values of fields.

Anything else that you want to happen when your page appears should be called from the `OnAppearing` method. A common use case is loading data, which will usually be an asynchronous operation as the data is loaded either from a database or from a web API, and you can't have an asynchronous constructor.

In `MauiTodo`, we wrote our own `Initialise` method to do this, but the logic in this method is an ideal candidate for the `OnAppearing` method. Open the `MauiTodo` app and edit the `MainPage.xaml.cs` file. In here, override the `OnAppearing` method and move in the logic from the `Initialise` method. Once that's done, you can delete the `Initialise` method, and the call to it from the constructor.

Keep the call to `base.OnAppearing()`, and make the method `async`. Listing

7.5 shows the new `OnAppearing` method.

Listing 7.5 The `OnAppearing` method in `MainPage.xaml.cs`

```
protected override async void OnAppearing() #A
{
    base.OnAppearing();#B

    var todos = await _database.GetTodos();#C

    foreach (var todo in todos)#C
    {
        Todos.Add(todo);#C
    }
}
```

Not shown in Listing 7.5 is the removal of the `Initialise` method, and the call to it in the constructor (`_ = Initialise()`), so remember to delete these.

You can run the MauiTodo app now and see that it still loads the to-do items from the database when the page appears.

7.3.2 `OnDisappearing`

As the name suggests, the `OnDisappearing` method is called when the page disappears. This can be useful for pages that manage state of some kind, for unsubscribing to events and messages, and for cleaning up resources that are no longer needed when the page is no longer visible.

7.3.3 `OnNavigatedTo` and `OnNavigatedFrom`

The `OnNavigatedTo` and `OnNavigatedFrom` methods perform essentially the same function as the `OnAppearing` and `OnDisappearing` methods respectively, but with one key difference. The `OnNavigatedTo` and `OnNavigatedFrom` methods are called before the page appears or disappears, whereas the `OnAppearing` and `OnDisappearing` methods are called after.

As we've seen, the `OnAppearing` method is a good option for loading data. Once the page is visible, you can show an `ActivityIndicator` (or a custom

loading indicator of your own) to display to the user that something is happening in the background. If you were to perform this action in the `OnNavigatedTo` method, the app would appear to stall while waiting for the data to load.

You will likely find that using the `OnAppearing` and `OnDisappearing` methods are the best fit for most use cases. But in scenarios where you need a quick, synchronous operations before a page appears or disappears, `OnNavigatedTo` and `OnNavigatedFrom` will be the better choice.

7.3.4 OnSizeAllocated

The `OnSizeAllocated` method is called whenever the size of the page changes. This can include when the page first loads, when changing orientation, or when a window is resized. The method takes two parameters of type `double` for width and height.

This method is useful for building responsive UI and resizing or repositioning elements in response to page size changes. For example, we saw in the previous chapter how `AbsoluteLayout` can use proportional positioning to place a FAB 9/10 of the way down and across the screen or window. However, using `OnSizeAllocated` lets you write more sophisticated rules about the position and size of your views. For instance, you might like to change both the size and margin of the FAB at certain breakpoints. This would differ from using proportional size and positioning, as you could write some logic to determine which of a set fixed sizes and positions to apply based on a range of screen or window sizes.

You can see an example using this approach in this chapter's code folder.

7.3.5 BackButtonPressed

The `BackButtonPressed` method is called when either the back button on the .NET MAUI navigation bar is used (you'll see this in action later in the chapter) or, if using Android, the Android OS back button is used. This can be useful if you want to execute some code when this event occurs.

NOTE

The `BackPressed` method should be used for supplementing, rather than changing, the behaviour of the back button. If you want to change the behaviour of the back button for Android users, you should do this by overriding the `OnBackPressed` method in the `MainActivity` class in the Android platform folder.

Unlike the other lifecycle methods mentioned here, which are all void, `BackPressed` has a return type of `bool`. However, as the method is called by an event handler, the return type is moot. You can't for example, return `false` instead of returning a call to the base method (which would be the default behaviour) to cancel navigation.

7.4 Navigation in .NET MAUI

Some apps work well with just a single page, but they tend to get more interesting when they include enough functionality to warrant multiple pages. The apps we've built so far have all been single page apps, and that's worked well for them. Throughout most of the remainder of this book, we'll be building MauiStockTake, a more complex app that will need multiple pages and, consequently, navigation. MauiStockTake will also rely on some of the more advanced concepts we'll learn throughout the rest of the book.

.NET MAUI supports three main navigation paradigms: hierarchical navigation, tabbed navigation, and flyout navigation. Each of these is supported by a subclass of `Page`, specifically configured to support each navigation paradigm. Additionally, .NET MAUI has `Shell`, which supports tabbed and flyout navigation as well as route-based navigation. Route-based navigation is especially helpful if you want to support deep linking (letting the host OS navigate to specific parts of your app using a unique URL).

Throughout the remainder of this chapter, we'll get an overview of MauiStockTake, the app that we'll use to build our .NET MAUI skills throughout most of the remainder of the book, and look at the way .NET MAUI supports the navigation that we will need to build a multi-page app.

7.4.1 Scenario: The MauiStockTake app

Mildred is the owner of Mildred's Surf Shack, a popular surf store near the beach. Mildred sells surfboards and surfing supplies, and hires surfboards to tourists. Mildred uses her sales figures every week to place orders with her suppliers to make sure she always has enough stock of consumables like wax. She also takes an inventory of her stock once a quarter to ensure that her formula is working well.

Once a quarter Mildred asks two of her staff members to come in overnight to help her take inventory. Each one uses a clipboard and writes their name at the top, and records what they see. The process works but has some problems. Firstly, the way people note down inventory is not consistent, and this compounds the second problem which is that it's difficult and time consuming for Mildred to reconcile hers and her staff's notes at the end. Finally, the process takes all night, and Mildred's staff don't like working all night any more than she likes paying them overtime.

As the owner of Beach Bytes, an app development company conveniently situated next door to Mildred's Surf Shack, Mildred thinks you can help her by building an app to help make her stock-taking process more efficient.

7.4.2 Features of the MauiStockTake app

The MauiStockTake app is concerned with three main areas:

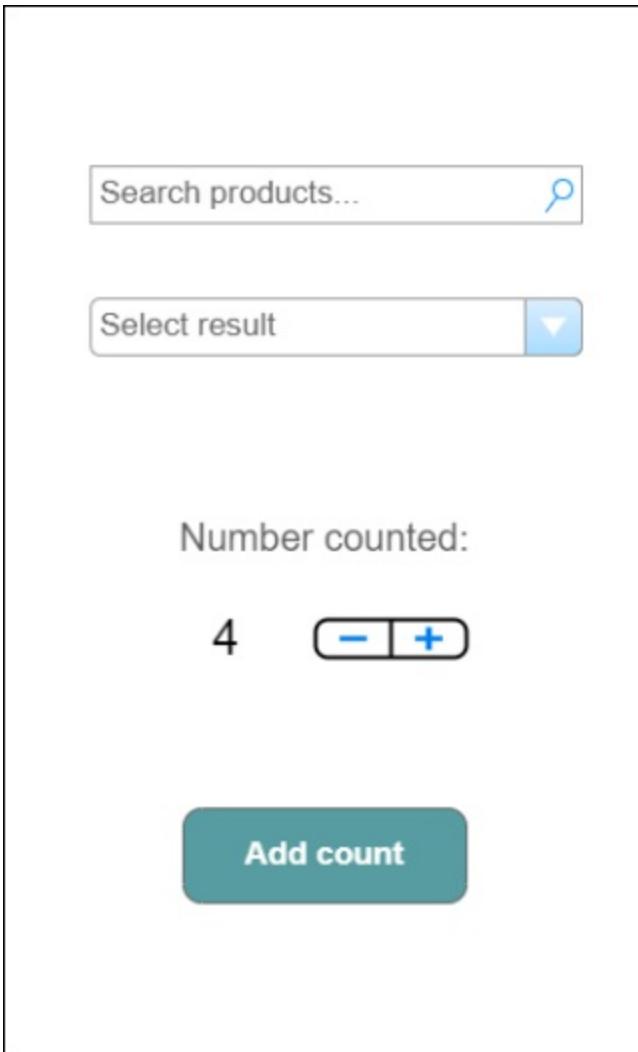
Table 7.1 A summary of the problem domain for the MauiStockTake app

Problem area	Description
Products	Staff record product names inconsistently, and sometimes get them wrong. To avoid this, Mildred wants the app to look up products in her catalogue rather than letting her staff enter details arbitrarily. She also wants us report on the manufacturers so that

	she can identify where she spends the most money on inventory and, hopefully, negotiate bulk discounts.
Inventory	This is the core behaviour of the app. She needs to take inventory of how many of each product she has.
Staff	Mildred wants to divide up her work into three areas and assign each area to a member of the team (including herself). She needs to know who is recording what, to simplify tracking down any errors. Additionally, when there is a large volume of stock to get through, she might want to rotate the team so that each area is inventoried by at least two people to catch any mistakes early.

Now that we understand the problem, we can start thinking about the solution. Each of these areas will need a page in the app that addresses the problem. Figure 7.6 shows a mock-up for a products page that would address the products problem area.

Figure 7.6 The product page. Includes a search field, and the results populate a dropdown, a stepper to indicate how many items were counted, and an Add button to add this count to the stock take.



The product page will be the heart of the app. We will have a lookup field where Mildred's staff can enter a product name and select the appropriate result from the list. Based on this we know we need a product lookup feature.

Figure 7.7 shows a login page mock-up, which would address the staff problem area.

Figure 7.7 The MauiStockTake app will need a login page

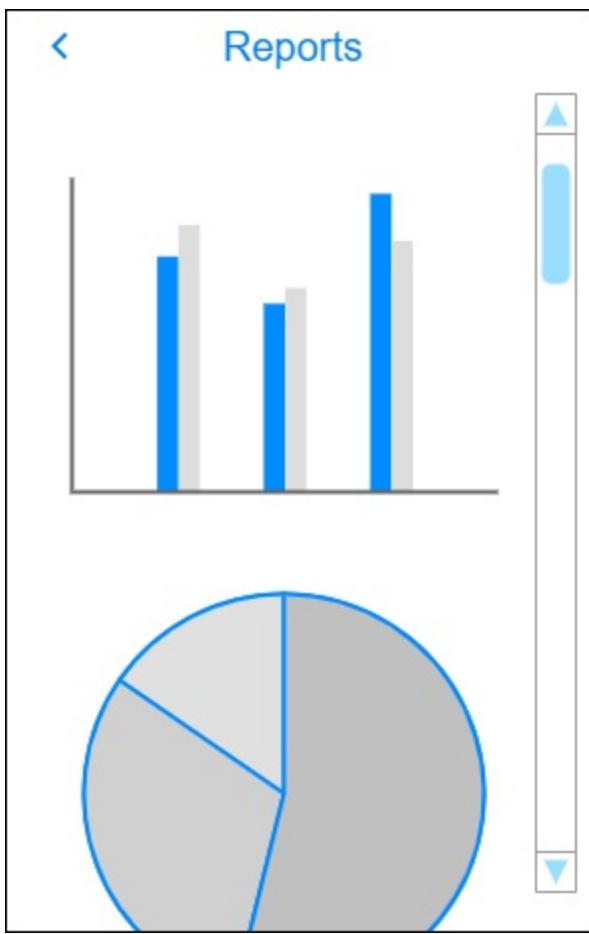
The image shows a simple sign-in form. At the top, it says "Sign In". Below that is a "User Name:" label followed by a text input field containing "johndoe". Underneath is a "Password:" label followed by a text input field containing "*****". At the bottom is a green "SIGN IN" button.

Mildred said she wants to be able to identify who has entered each inventory record, so we need a way for staff to log in to the app. Or, at least, enter their name. This means that we will need a user service so that the staff can identify themselves.

Finally, we need a way to review the results of the stock take on each device, and we'll also want to sync them to an upstream service so that they can be consolidated into a single comprehensive report. For both of these we'll need an inventory service, which will aggregate the inventory counts as they come in, list the results for the user, and sync them to the upstream service.

Figure 7.8 shows a mock-up for a report page, that would be used to view the aggregated stock counts.

Figure 7.8 The MauiStockTake app will need a reports page.

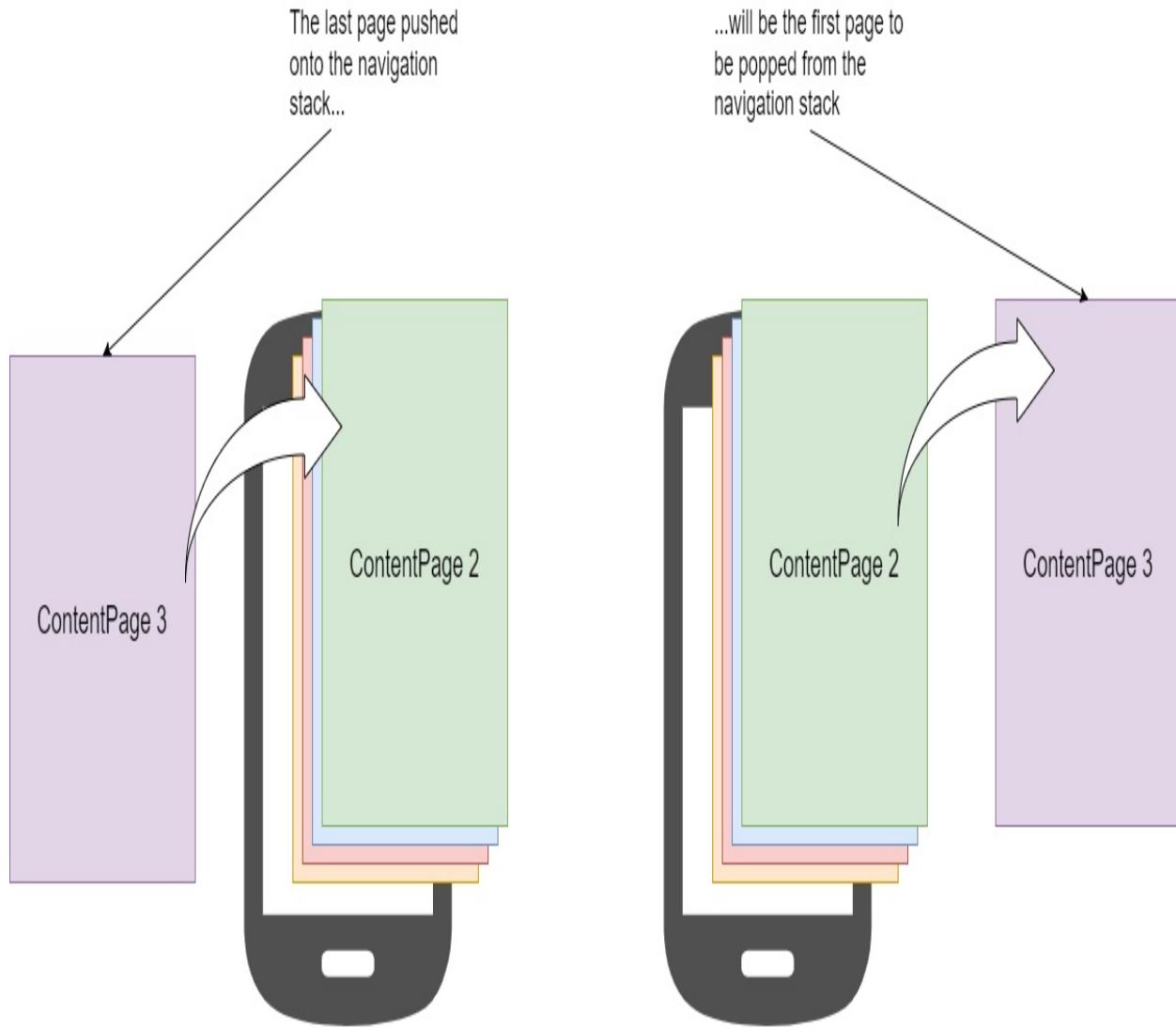


In order to accommodate the multiple pages in the app, we'll need to adopt a navigation paradigm so that users can move between the different pages.

7.4.3 Hierarchical Navigation

In .NET MAUI, `NavigationPage` is used to provide *hierarchical navigation* by showing `ContentPages` in a navigation stack. In this case, the term ‘stack’ has its standard definition, meaning it allows elements to be pushed (added) onto the stack and popped (removed) from the stack. In the case of a `NavigationPage`, the elements that are pushed or popped from the stack are Pages.

Figure 7.9 In hierarchical navigation, a page can be pushed onto the stack (left), or popped from the stack (right), following a last-in, first-out pattern.



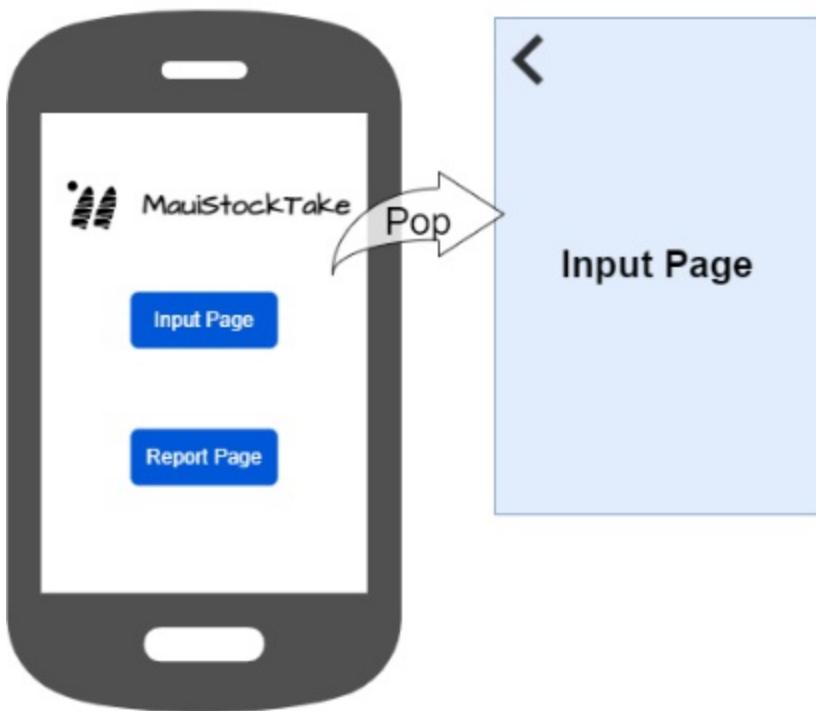
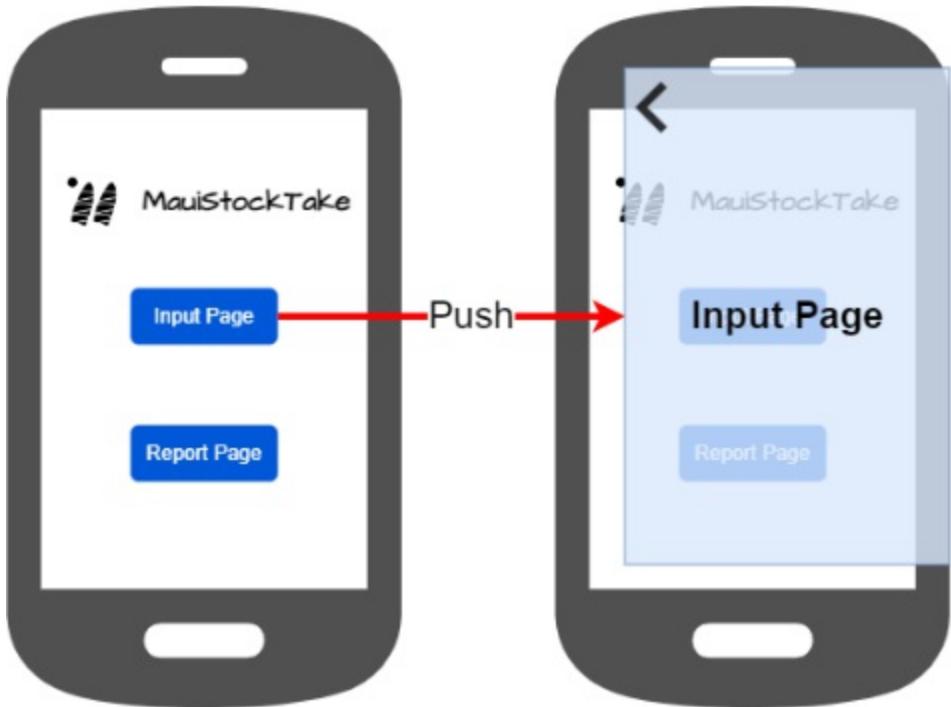
The most recently pushed (or last-in) page to the stack is the page that your user will see on screen. Any class that is derived from the `Page` base class can be pushed to a navigation stack, including any of the page types listed in this section. With this approach you can combine navigation paradigms, although this can be confusing for your users so be careful not to use unfamiliar combinations.

Hierarchical navigation is a good option if you want your app to start with a menu screen. We could use this approach for MauiStockTake by starting with a menu page, offering buttons to navigate to the input page and report page. Tapping one of the buttons would push the relevant page onto the stack. When you use `NavigationPage`, a navigation bar is shown at the top that has

a back button, which pops the current page to take you back one step in the stack.

Figure 7.10 shows how this could work with MauiStockTake.

Figure 7.10 MauiStockTake with hierarchical navigation. The app starts with a menu page, offering two buttons that correspond to the available pages in the app. Tapping one of these pushes the corresponding page onto the navigation stack. Tapping the back button (the chevron in the top left) within a page pops the page from the stack, returning you to the previous page.



Hierarchical navigation differs from the other navigation paradigms in one key way. With the other paradigms pages are bound to a UI control like a tab or a flyout item and tapping or clicking on them will result in the page being displayed automatically by the framework. With hierarchical navigation, you

push pages onto the navigation stack programmatically.

In figure 7.10 we use a Button to navigate to a page. If we were implementing this in code, we would use the PushAsync method on the Navigation class to achieve this, passing in an instance of the page type we want to navigate to as a parameter:

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

To pop the page from the stack, and navigate back to the previous page, we would use the PopAsync method:

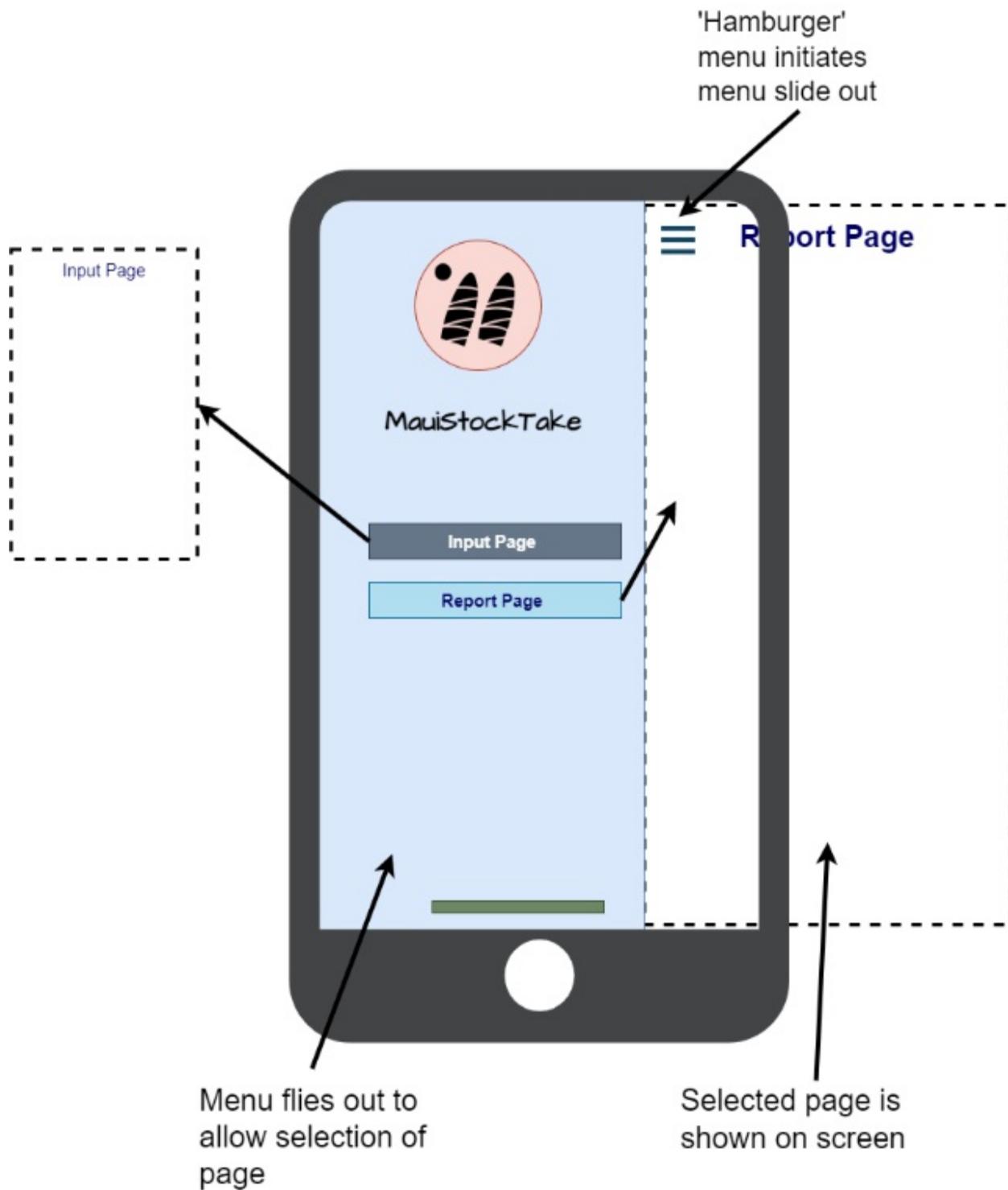
```
await Navigation.PopAsync();
```

While we won't use it in this book, hierarchical navigation is very popular, and is likely what you will use if you have a menu page or landing page in your app.

7.4.4 Flyout Navigation

With flyout navigation, a menu slides (or "flies") out from the side of the screen and presents a list of pages which can be displayed in the main display area. The user selects a page, which then occupies the main display area while the menu slides back away. Figure 7.11 shows how we could use this navigation paradigm for MauiStockTake.

Figure 7.11 In a FlyoutPage, a menu (or Flyout) displays a list of pages, and the selected page is displayed on the screen



In .NET MAUI, `FlyoutPage` is used to support the flyout navigation paradigm. `FlyoutPage` has two child properties of type `Page`, one called `Flyout` which is used for the menu, and one called `Detail` which the selected page is assigned to and displayed on screen.

You can use a `ContentPage` to build a fully customised flyout menu and assign it to the `Flyout` property of `FlyoutPage`. This gives you a lot of flexibility, although you should be cautious of introducing unfamiliar UX paradigms to your users.

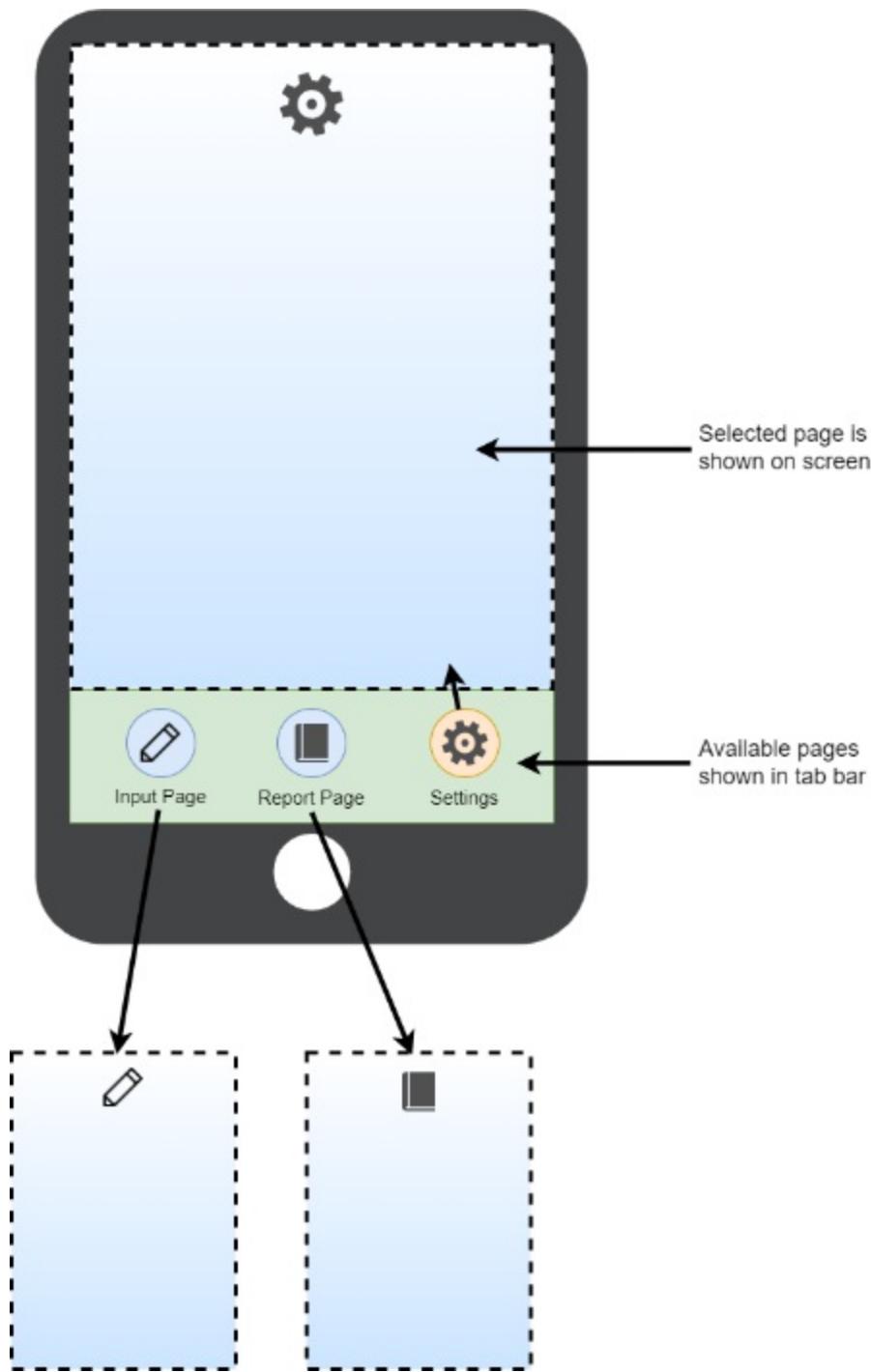
Within your `FlyoutPage`, you will need to supply a collection of type `FlyoutPageItem` that your user can select from. `FlyoutPageItem` has a property called `TargetType`, which you assign the desired page to. When your user selects one of these `FlyoutPageItem`, the selected item's `TargetType` property will be assigned to the `Detail` property of the `FlyoutPage`, causing it to be displayed in the main display area of your app.

7.4.5 Tabbed Navigation

With tabbed navigation, tabs are used to provide access to the various pages available within the app. A `TabBar` is displayed with a collection of labelled icons, each one representing a page in the app that the user can select. The selected page is displayed in the main display area. By default, on iOS the `TabBar` will be displayed along the bottom of the screen. On Windows and Android, the `TabBar` is displayed at the top, more closely resembling the tabs you're used to seeing in web browsers.

Figure 7.12 shows how we could use this navigation paradigm in `MauiStockTake`.

Figure 7.12 A TabBar is displayed along the bottom of the screen showing a collection of labelled icons, each representing a page in the app. The user can tap on one of these icons, and the page that it represents will be displayed on screen.



‘Tab’ is a skeuomorphic metaphor that doesn’t always hold up (much like the floppy disk icon), as it’s rare to see actual tabs now, with most UI designs favouring simple icons instead and displaying them in a bar at the bottom, similar to the default iOS approach.

In .NET MAUI, `TabPage` enables the tabbed navigation paradigm. Unlike

ContentPage, which has a single child Content property, TabbedPage allows multiple children, each of which must derive from Page. These pages are automatically displayed as tabs using the platform default approach, with the icon and text taken from the Page's Icon and Title properties respectively.

7.5 Introducing Shell

In .NET MAUI apps, Shell is a simple way to describe the page navigation hierarchy of your app in XAML. It provides a simple way to structure your whole app, using flyouts and tabs to navigate around.

If your app design is based on flyout navigation or tabbed navigation (or both), Shell can be a better choice than using the page types that support those paradigms. Shell provides three chief advantages:

- **Whole app mapped in one file.** The Shell file itself is an easy one-stop-shop to see how your whole app is organized navigationally. It's easy to see how the pages of your app fit together, and equally easy to shuffle them around if needed.
- **Dependency Resolution.** We've already seen how .NET MAUI uses the generic host builder pattern, which includes .NET's built-in DI container. Using Shell, any dependencies that are constructor-injected into your pages will be automatically resolved for you, just like Controller dependencies in an ASP.NET Core app.
- **Route-based navigation.** With Shell, each page in your app is reachable via a URL, and this provides two cool features. The first is that it simplifies deep linking, enabling your app to be opened externally to a specific page (especially useful for push notifications). The second is that you can use query parameters, meaning you can pass data to your pages as part of the URL.

Complex Query Parameters in .NET MAUI

Shell was originally introduced in Xamarin.Forms version 4 and included query parameters. If you've previously used Shell in Xamarin.Forms, you'll be pleased to see a significant improvement in .NET MAUI.

In Xamarin.Forms, Shell query parameters were limited to primitive types, so you could easily pass a `string`, `int` or `bool` for example. But you couldn't pass a more complex object (unless you used a workaround like serialising it to JSON). If you wanted to route, say, from a product list page to a product details page, you would need to pass the id of the product, and your details page (or its ViewModel) would be responsible for looking up the details of the product.

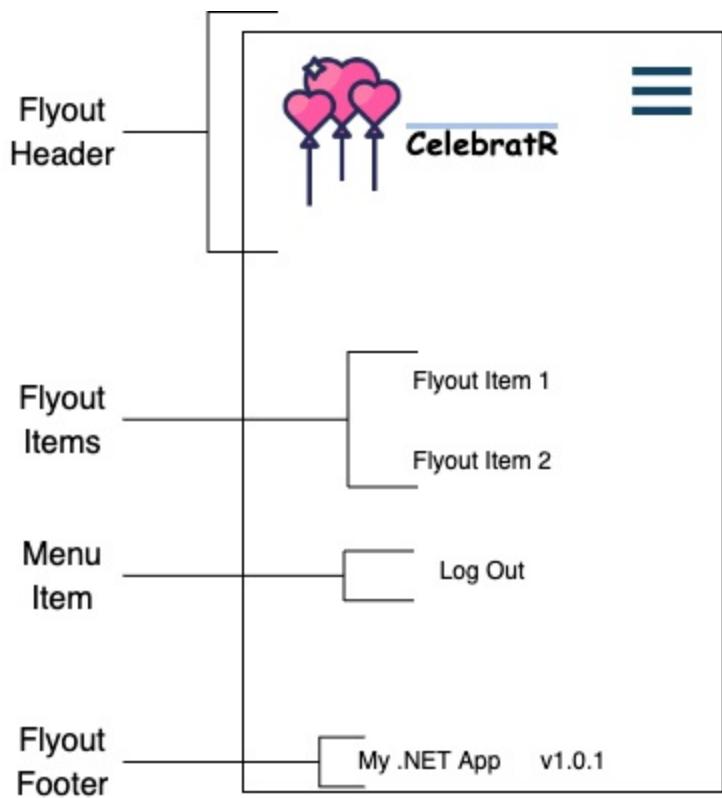
With .NET MAUI, Shell lets you pass objects directly as a parameter so, in the above example, you could route to your product details page and pass the whole product.

Shell lets you combine the flyout and tabbed navigation paradigms in the way that best suits your app.

7.5.1 The Flyout Menu

Shell provides a flyout menu that can be used for navigation as well as providing access to features or functionality that you would expect anywhere across your app. It features a customizable header and footer, and lets you add flyout items for navigation, and menu items to execute functionality.

Figure 7.13 The flyout menu in Shell consists of a customizable header and footer, flyout items for navigating the app, and menu items for executing code.



The header is easy to customize. You can write XAML UI directly inside the `AppShell.xaml` file or you can use a control template. You can also write your XAML UI directly inside Shell for your footer as well. Or, you can use control templates or imported views.

Menu Items

The flyout menu in Shell supports menu items as well as flyout items, meaning you can assign functionality to the items in the flyout rather than having them strictly navigate to a page in the app (think, for example, of a logout button).

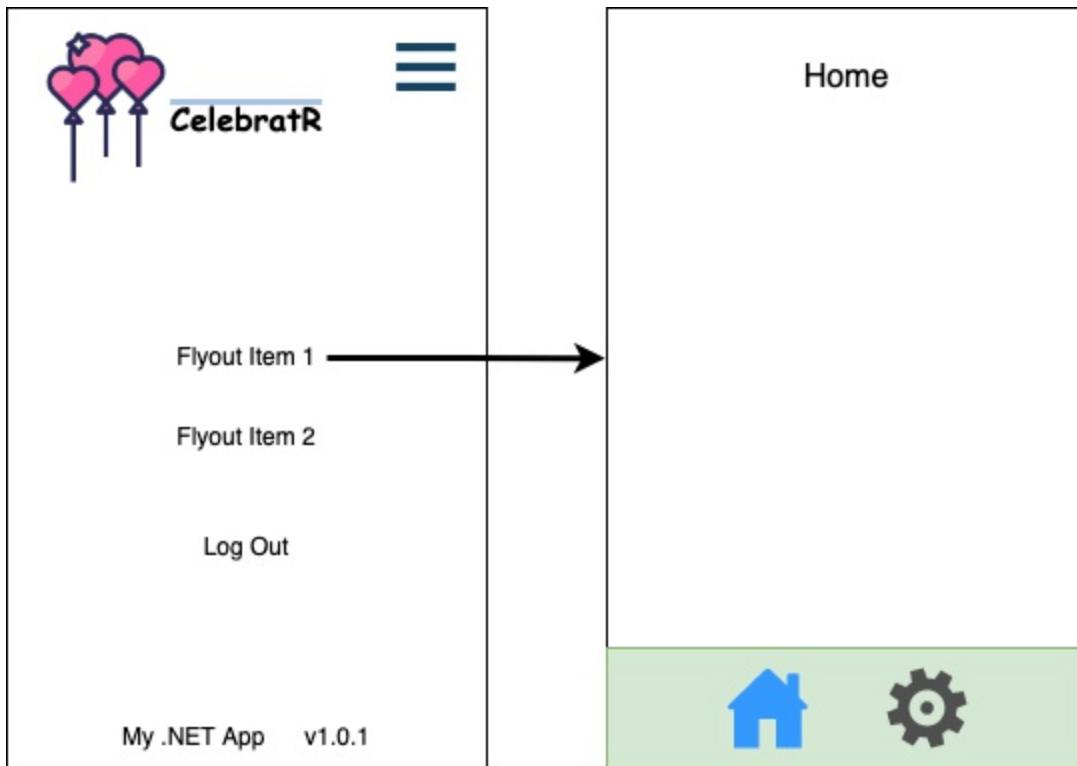
Figure 7.13 shows a Log Out button added to the Shell flyout menu. This is a menu item as it is used to execute some code (the logic to log the user out of the app) rather than to navigate to a page within the app.

The `MenuItem` type has a `Clicked` event handler and a `Command`, both of which can be used to call the code you want to execute when the user taps or clicks the menu item.

Flyout Items

Flyout items provide access to pages or groups of pages within your app. The `FlyoutItem` type allows multiple children, meaning you can add a single page or a collection of pages.

Figure 7.14 A `FlyoutItem` in Shell is configured to show a section of the app. This section contains two tabs: a Home tab and a Settings tab. Users can switch between the two within this Shell section. Opening the menu reveals other flyout items, allowing navigation to other Shell sections.



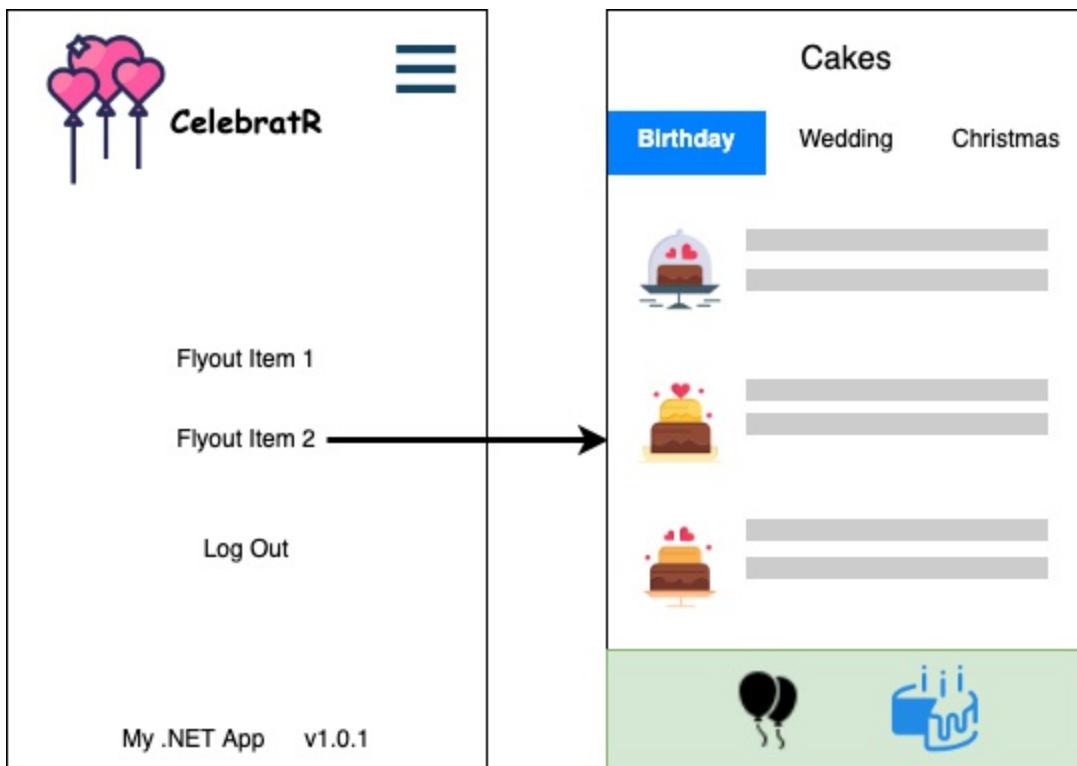
With Shell, your app is divided into Shell Sections. Each Shell Section can contain one or more pages or can contain collections of pages organised by tabs.

7.5.2 Tabs

Tabs are organised hierarchically in Shell. Unlike with `TabbedPage`, when using Shell, the tabs at the highest tier of the hierarchy (i.e., those that you add first) will always appear at the bottom of the screen, no matter the platform. Pages can be embedded within a tab, which will then themselves be

individually tabbed. Tabs at this level of the hierarchy are shown at the top of the screen.

Figure 7.15 Flyout Item 2 navigates to a Shell Section that has two tabs, accessible via a Balloons icon and a Cakes icon, both displayed at the bottom of the screen. Within the Cakes tab are three additional tabs, providing access to content at the next tier down. These tabs are Birthday, Wedding and Christmas, accessible at the top of the screen.



For large and complex applications, this ability to partition the app into sections with Shell can be a powerful tool to logically organise your app's content while still ensuring users can easily navigate to the parts they need.

7.5.3 Getting Started with MauiStockTake

We're ready to get started building our big project! We'll continue building MauiStockTake throughout the rest of this book as we learn more about services, architecture and design patterns. We're going to get started with it in this chapter as we learn about navigation, and use Shell to build the structure of the app.

Download the Starter Project

The MauiStockTake app will need a back-end API to query for products and to send inventory counts to. At Beach Bytes, one of your colleagues will focus on this aspect of the solution, allowing you to concentrate on the client app.

A prototype of the MauiStockTake API is available for you to download from the book's online resources.

Download the API and have a look at the solution. The solution is based on Jason Taylor's Clean Architecture (CA) template. If you are interested you can find out more about it at the GitHub repository, which you can access using this link: <https://tinyurl.com/2p9ceh4x>. I use Jason's CA approach regularly and sometimes help him teach it, so I recommend spending some time learning it if you can. But it's well outside the scope of this book, so for now just run the solution.

Once you've confirmed that the solution successfully runs, you can move on to adding the .NET MAUI project.

Add the .NET MAUI App

Up until now we've been using the `blankmaui` template that I gave you. The reason for using this template is that the default template has Shell built in. You can work with this even when not using Shell, but I think it's easier to learn some fundamental principles of .NET MAUI before we start muddying the waters with Shell.

Should I use Shell for every app?

Shell is a good fit for apps using flyout or tabbed navigation, or both, but for some scenarios it's not the best option.

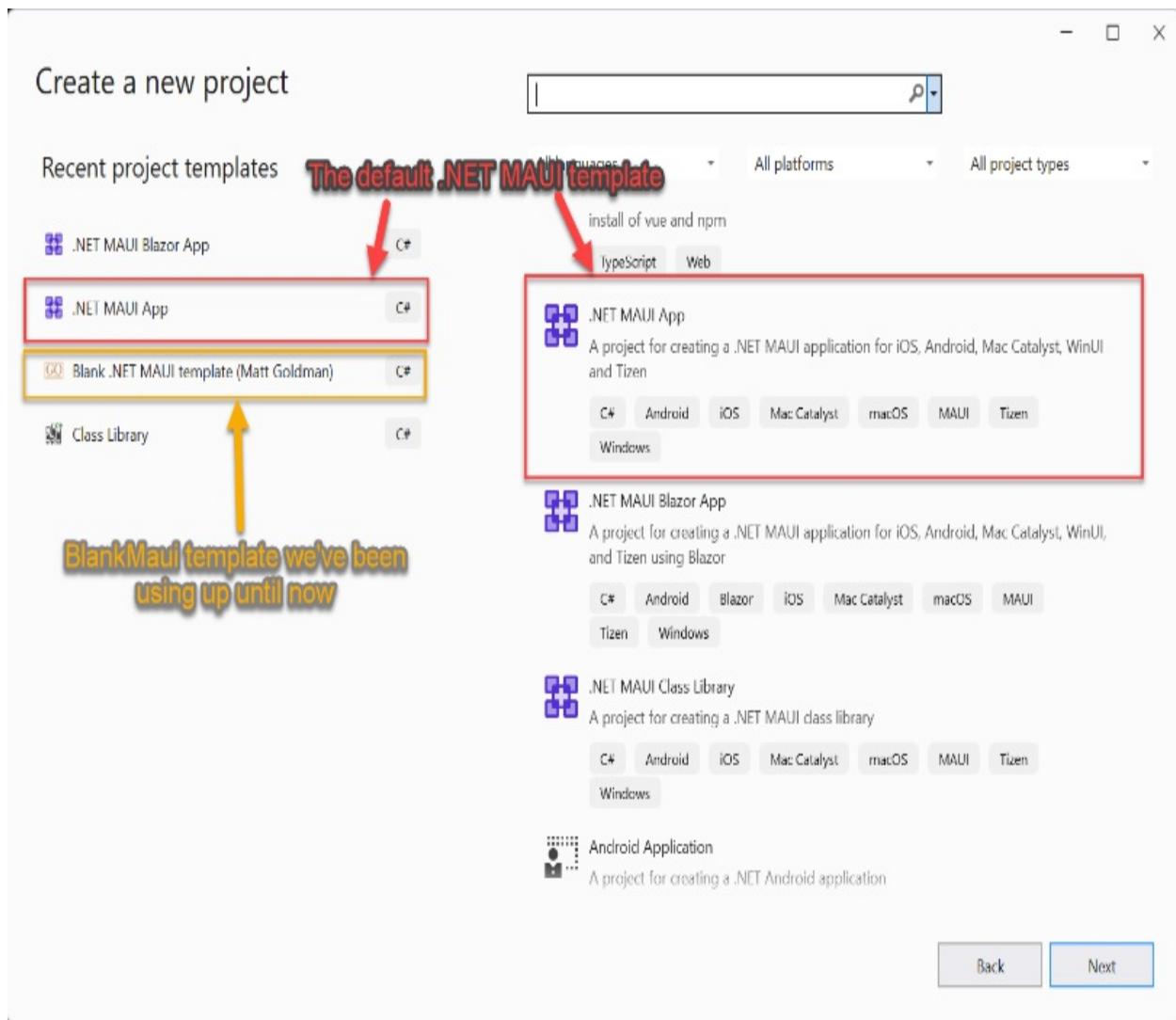
Line of business desktop applications, for example, don't typically feature this style of navigation, which is more traditionally associated with web or mobile apps. In a single page application, like those we've been building so

far in this book, or apps using hierarchical navigation, Shell is unnecessary. Finally, Shell lets you extensively customize the flyout, but you can't customize the tab bar at all, so if your visual design isn't accommodated by what Shell provides, you need to go with a different option.

For many apps though, including MauiStockTake, Shell is perfect.

MauiStockTake will use both tabs and a flyout, and Shell will make it easy for us to build, so this time, we're going to use the default template. Add a new .NET MAUI project to the MauiStockTake solution and call it `MauiStockTake.UI`. The default template is just called .NET MAUI App. If using Visual Studio, the template will look similar to figure 7.16.

Figure 7.16 The default .NET MAUI App template includes Shell out of the box. The blankmaui template we've been using up until now is exactly the same, but without Shell.



This template will create a new .NET MAUI app almost identical at this stage to the Aloha, World project. Have a look in solution explorer and you will see some key differences.

The first is that the project includes `AppShell.xaml` and `AppShell.xaml.cs` files. These are the Shell files that define the structure and hierarchy of your app. We'll come back to these shortly.

The next change is how the app bootstraps our main page. In Aloha, World, and all the projects we've built so far, the `MainPage` property in `App.xaml.cs` has been assigned the main page of our app. If you look at `App.xaml.cs` in `MauiStockTake.UI`, you'll see that a new instance of the `AppShell` class is assigned to it instead.

If you open `AppShell.xaml.cs`, you'll see that the root node is of type `Shell` and has a number of attributes we're familiar with from `ContentPage`. You'll see that it has one child element of type `ShellContent`. This `ShellContent` instance has the following three properties assigned:

- **Title:** This property sets the page title. The title is displayed in the navigation bar when the page is active, and in the tab bar when the page is available as a tab.
- **ContentTemplate:** This is the actual content to be displayed. This is of type `DataTemplate`, so you could build content directly in your `AppShell.xaml` file to allocate to it if you wanted. It's more efficient to assign a `ContentPage` to this property, as has been done here, as the page will only be loaded when navigated to. Once you start factoring a page's dependencies, as we'll see in the next chapter, this can add up to a lot of overhead. In the template, the `MainPage` type has been assigned from the local namespace, and if you look at the `Shell` attributes, you'll see that the `MauiStockTake.UI` root namespace has been added to the local XML namespace, which is where the `MainPage` class is located.
- **Route:** This particular `Shell` content has been assigned the route `MainPage`. This means that when we navigate with `Shell`, we can easily get to this page using this route. We'll look at routes and navigation shortly.

We're not going to use the `MainPage.xaml` file in our app, so we need to replace this item in the `Shell` with the pages we want to use. We already know what pages we need in the `MauiStockTake` app, even though we haven't finalised the visual or functional designs. But just knowing the pages is enough to get started building the `Shell`.

We'll add the pages shortly and then get the `Shell` structure completed. But before we do, there's one last step in our project setup that's going to make things easier for us as we go. C# 10 introduced global using statements, which allow us to mark a using statement in any file as global, which then makes the namespace it imports available to any other file in the same project. We can make use of these in .NET MAUI projects to reduce boilerplate code referencing namespaces that we will repeatedly use across different files.

Add a file called GlobalUsings.cs to the root of the project. Depending on how you created this file, there may be some templated code in there. If that's the case, delete it. We'll leave the file blank for now and add namespaces to it as we go.

Our solution is now set up and ready for us to start fleshing out the features that Mildred needs in the MauiStockTake app.

Add the InputPage and ReportPage

We've got enough of our high-level design in place to start building our app. The app has two functional pages (input and reports), as well as the login page, but this will only be used when the user first launches the app. A two-page app lends itself well to a tab bar, but we'll also add a flyout menu that can show some information about the app and the user and provide a logout button.

We know we're going to need a login page, input page, and reports page, so let's add these now. Create a folder in the `MauiStockTake.UI` project called `Pages`, and add a .NET MAUI ContentPage (XAML) called `InputPage`, another called `LoginPage` and a third called `ReportPage` to this folder.

Remove all the content from each page. In the `InputPage` and `ReportPage`, add a `VerticalStackLayout` with a single `Label`, with the `Text` property set to "Input Page" and "Report Page" respectively. Do the same with the `LoginPage`, but add a button underneath the `Label` that has "Login" as the `Text` property. The XAML for each of these pages is in listings 7.8 to 7.10.

Listing 7.8 InputPage.xaml

```
<?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="MauiStockTake.UI.Pages.InputPage"
    Title="InputPage">
    <VerticalStackLayout>
        <Label Text="Login Page"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="CenterAndExpand" />
```

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

Listing 7.9 LoginPage.xaml

```
<?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="MauiStockTake.UI.Pages.InputPage"
              Title="InputPage">

    <VerticalStackLayout>
        <Label Text="Input Page"
              VerticalOptions="CenterAndExpand"
              HorizontalOptions="CenterAndExpand" />
    </VerticalStackLayout>
</ContentPage>
```

Listing 7.10 ReportPage.xaml

```
<?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="MauiStockTake.UI.ReportPage"
              Title="ReportPage">

    <VerticalStackLayout>
        <Label Text="Report Page"
              VerticalOptions="CenterAndExpand"
              HorizontalOptions="CenterAndExpand" />
    </VerticalStackLayout>
</ContentPage>
```

We're going to refer to these pages in a few places throughout our app, so add a global using statement for the namespace to the `GlobalUsings.cs` file:

```
global using MauiStockTake.Client.Pages;
```

We're almost ready to add these pages to the Shell, but before we do, we need to import the icons we're going to use for these pages in the tab bar.

Import Required Assets

We'll need an icon for each of the pages in our tab bar, as well as some other menu item icons for the flyout, as well as a header image. In the book's online resources for this chapter, you'll find five images:

- icon_input.svg
- icon_login.svg
- icon_logout.svg
- icon_report.svg
- surfshack_logo.jpeg

Download these images and copy them into the Resources/Images folder of the MauiStockTake.UI project. Now that we've got the Tab icons, let's start building out the tab bar and add the pages to the Shell.

Add the TabBar

Before we add the tabs for our pages to the app, let's get rid of the boilerplate code that we're not using. Open AppShell.xaml, and delete the existing ShellContent:

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

We're not going to use this MainPage in our app, so we can delete the MainPage.xaml and MainPage.xaml.cs files as well. Now that we've got a clean slate, we can add the tab bar. Start by adding this now:

```
<?xml version="1.0" encoding="UTF-8" ?>  
<Shell ...>  
  
    <TabBar>  
  
        </TabBar>  
  
</Shell>
```

We want the tab bar to display tabs for the InputPage and the ReportPage, so we'll add a Tab for each of these, and for each Tab, we'll assign a Title and

Icon.

Remember, as we saw in section 7.5.2, that a Tab can have multiple ShellContent children, which would display top tabs within the bottom tab for each page. We're not going to do this though; we'll add one ShellContent child to each Tab and assign a page to the ContentTemplate property. To do this we will need to bring in an XML namespace to represent the Pages namespace in our app.

Listing 7.11 shows the code for AppShell.xaml. Add the TabBar and the Tabs to make your code match the listing.

Listing 7.11 AppShell.xaml

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

    <TabBar>
        <Tab Title="Input" #A
            Icon="icon_input.svg">#B
            <ShellContent ContentTemplate="{DataTemplate Pages:In
        </Tab>
        <Tab Title="Reports"
            Icon="icon_report.svg">
            <ShellContent ContentTemplate="{DataTemplate Pages:Re
        </Tab>
    </TabBar>

</Shell>
```

At this point the high-level visual structure of our app is defined. Launch the app and confirm that it runs. You should see something like figure 7.17.

Figure 7.17 The shell of the MauiStockTake app running on Android. At this stage the InputPage and ReportPage don't do anything other than display a label with their name. But you can tab between them to verify that the pages are created correctly, and the routes are registered.



Verify that the icons are displayed and that you can tab between the two pages. The label and title should change depending on which page you're on. Once you're happy, delete the `MainPage.xaml` and `MainPage.xaml.cs` files as we won't be using them anymore.

Add the Flyout

We've got a TabBar now which is enough for the user to navigate between the two main pages in the app once they're logged in. But we also want a Flyout, we won't use the Flyout for navigation, but we'll use it to provide a way to log out of the app and show a header to give the user some information about the app.

The first thing we will want to do is enable the Flyout. To do this, set the FlyoutBehavior property of the Shell to Flyout:

```
<Shell  
    x:Class="MauiStockTake.Maui.AppShell"  
...  
    FlyoutBehavior="Flyout">
```

Next, we will want to add a `MenuItem` to the `Shell`, set the `Text` property to “Logout”, and the `IconImageSource` property to the filename of the logout icon image that we imported.

```
<MenuItem Text="Logout"  
         IconImageSource="icon_logout.png" />
```

`MenuItem` has both an event and a `Command` that we could wire up, but we don't need this functionality just yet. If you run the app now, the `MenuItem` looks a bit janky, so we're going to specify the `Shell.MenuItemTemplate` so that we can tweak the layout a little. Inside this tag we can specify a `DataTemplate`, and in here we can add our layout, which is a process we've seen used for collections. Each `MenuItem` has an `Icon` and a `Text` property, and we can bind to these in our layout.

Listing 7.12 shows the code to add to AppShell.xaml. Most of the code has been omitted, and the code to add is shown in **bold**.

Listing 7.12 Shell.MenuItemTemplate

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell ...>

<Shell.MenuItemTemplate>
    <DataTemplate>
        <Grid ColumnDefinitions="0.2*,0.8*">
            <Image Source="{Binding Icon}"
                  Margin="35,0,0,0"
                  HeightRequest="45" />
            <Label Grid.Column="1"
                  Text="{Binding Text}"
                  Margin="10,0,0,0"
                  VerticalTextAlignment="Center" />
        </Grid>
    </DataTemplate>
</Shell.MenuItemTemplate>
```

```

        </Grid>
    </DataTemplate>
</Shell.MenuItemTemplate>

...

</Shell>

```

This isn't the most sophisticated layout; in fact, it's similar to the default layout, but a little more refined. You could be as creative as you like with these, and while we've only got one `MenuItem` in this app, if you have more, the template will apply to all of them.

Before we run the app, let's add the `Flyout` header. We've already got a feel for what a `Flyout` header can look like in the mockups we've seen earlier in the chapter (one for MauiStockTake and another for an app called `CelebratR`).

You can assign an arbitrary layout to `Shell.FlyoutHeader`, so building it will be straightforward. We'll use a `Grid` to position the controls, and show an `Image` with the app's logo, clipped to make it circular, and a `Label` underneath with the app's title.

Listing 7.13 shows the code for the `Flyout` header. Most of the code has been omitted, and the added code is in **bold**.

Listing 7.13 The Shell Flyout header

```

<?xml version="1.0" encoding="UTF-8" ?>
<Shell ...>

    <Shell.FlyoutHeader>
        <Grid RowDefinitions="*, *"
              Padding="20">
            <Image Source="surfshack_logo.jpeg"
                  WidthRequest="100"
                  HeightRequest="100"
                  HorizontalOptions="Center"
                  VerticalOptions="Center">
                <Image.Clip>
                    <EllipseGeometry Center="50, 50"
                                    RadiusX="50"

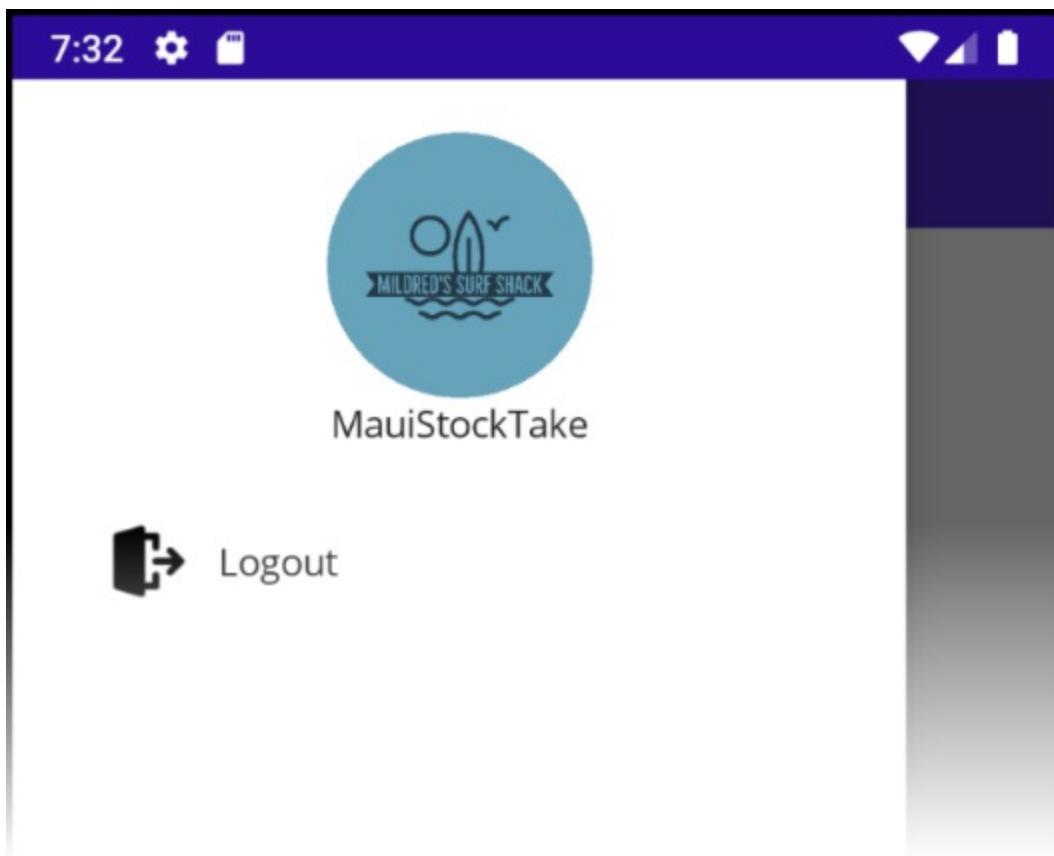
```

```
                    RadiusY="50" />
            </Image.Clip>
        </Image>
        <Label Grid.Row="1"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            Text="MauiStockTake" />
    </Grid>
</Shell.FlyoutHeader>

...
</Shell>
```

If you run the app now, you'll notice the hamburger menu in the top left corner. Tap or click on this and you'll see the Flyout, well, fly out. You should see something similar to figure 7.18.

Figure 7.18 The Flyout in MauiStockTake. At the top, you can see the Image which is clipped using EllipseGeometry, and underneath that is the Label with the apps' title. These two together (Image and Label) make up the Flyout header. Underneath those you can see the Logout menu item, which uses the imported icon and is styled using the Shell.MenuTemplate



7.5.4 Routes and Navigation

So far, we've got two pages in the app, and these are easily accessible via the tab bar. With Shell, most of your pages will be accessed in this fashion; either via a Tab or a Flyout item. While simplified navigation in this way is the default behaviour with Shell (and the main reason why Shell was introduced), it is still possible to navigate programmatically if you need to.

To navigate programmatically with Shell, we use the `GoToAsync` method:

```
await Shell.Current.GoToAsync("myroute");
```

With Shell, you use the `GoToAsync` method with a route, rather than an instance of a page (as in hierarchical navigation). In order to use the route, though, it needs to be registered, and this is easy when using `Tab` or `FlyoutItem` as they both have a `Route` property. Let's create a route for the two pages we have so far in our app:

```

<TabBar>
    <Tab Title="Input"
        Icon="icon_input.svg"
        Route="input">
        <ShellContent ContentTemplate="{DataTemplate Pages:Input}>
    </Tab>
    <Tab Title="Reports"
        Icon="icon_report.svg"
        Route="reports">
        <ShellContent ContentTemplate="{DataTemplate Pages:Report}>
    </Tab>
</TabBar>

```

The pages that we've registered routes for so far are easily accessed via the tab bar, so the routes aren't all that useful. But we have another page though that won't be accessible via either tabs or flyout, which is the login page.

As we don't have a tab or flyout item for the `LoginPage`, we will need to use the `GoToAsync` method to access it. And we'll also need a way to register a route for it, without specifying the `Route` property of a `Tab` or `FlyoutItem`. This is done with the `RegisterRoute` method on the `Routing` class. The easiest way to use this method is to pass two parameters: the first is a string to use for the route, and the second is the type of the page the route should represent.

We'll register a route for the `LoginPage` now. Open the `AppShell.xaml.cs` file, and register a route for the `LoginPage` in the constructor. Listing 7.14 shows the updated constructor for `AppShell.xaml.cs`.

Listing 7.14 the updated AppShell.xaml.cs constructor

```

public AppShell()
{
    InitializeComponent();
    Routing.RegisterRoute("login", typeof(LoginPage));
}

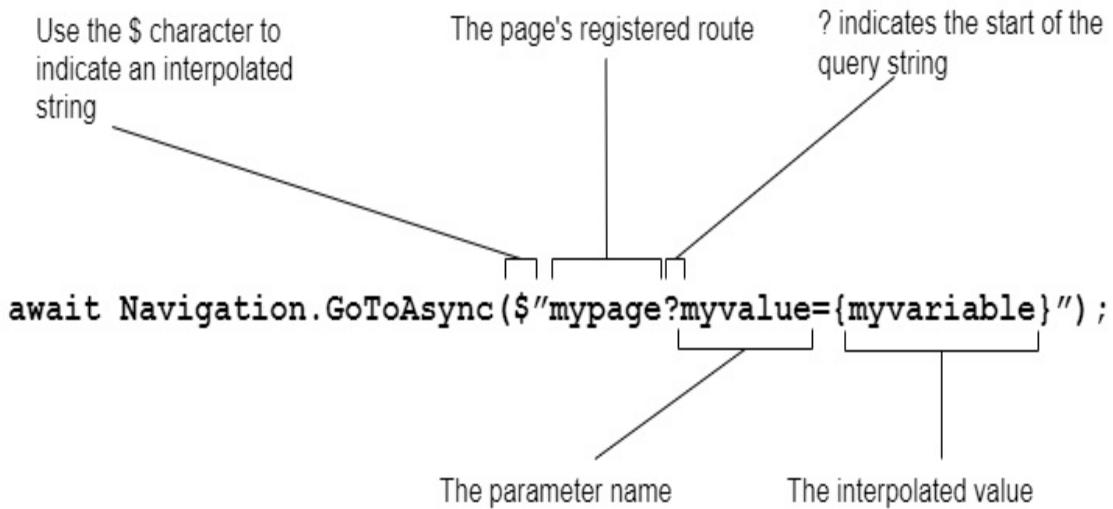
```

With this route registered, we can easily navigate to the `LoginPage`, even though it doesn't have a corresponding `Tab` or `FlyoutItem`, using the `GoToAsync` method. We'll revisit `LoginPage` in chapter 8 when we look at authentication.

7.5.5 Route Parameters

Routing in Shell is similar to web in that it allows you to pass query parameters. So rather than simply using `GoToAsync` with a route, you could pass values as well:

Figure 7.19 The `Navigation.GoToAsync` method with a route and parameter supplied



In the example in figure 7.19, string interpolation is used to pass the value of a string as a parameter to the `mypage` page. You don't have to use string interpolation; we could have hard-coded the value we wanted to pass in that URL. Also, you are not limited to strings. You can also pass navigation state, which is a dictionary of `<string, object>` and use the key in the receiving page (or the page's binding context) get the corresponding value from the dictionary.

To see how this works in practice, we can add a product details page to our app. First, let's add a class to represent products. In `MauiStockTake.UI`, add a `Models` folder, and in here add a new file called `Product.cs`. This can contain a class called `Product`, that will have four properties:

- `id` of type `int`
- `Name` of type `string`
- `ManufacturerId` of type `int`
- `ManufacturerName` of type `string`

Next, add a XAML ContentPage called ProductPage to the Pages folder. Then register a route for this page in the constructor of AppShell.xaml.cs, right under the line where we register the LoginPage route:

```
Routing.RegisterRoute("productdetails", typeof(ProductPage));
```

Later, we'll use this page to receive actual product information from our API and display details, but for now we'll just use it to test that our Shell and routing are set up correctly. We'll add a Button to the InputPage that says "Go to product", and wire it up to an event handler in the code behind file. In here, we'll instantiate a mock product that represents the app itself, add it to a dictionary, then navigate to the ProductPage, passing the dictionary as a navigation state parameter.

Listing 7.15 shows the updated content of the InputPage (the ContentPage tags have been omitted). The added code is in **bold**.

Listing 7.15 The updated InputPage content

```
<VerticalStackLayout Spacing="50">
    <Label
        Text="Input Page"
        VerticalOptions="Center"
        HorizontalOptions="Center" />

    <Button Text="Go to product"
        WidthRequest="200"
        Clicked="Button_Clicked"/>
</VerticalStackLayout>
```

Listing 7.16 shows the event handler to add to the InputPage.xaml.cs file.

Listing 7.16 the Go to product button event handler

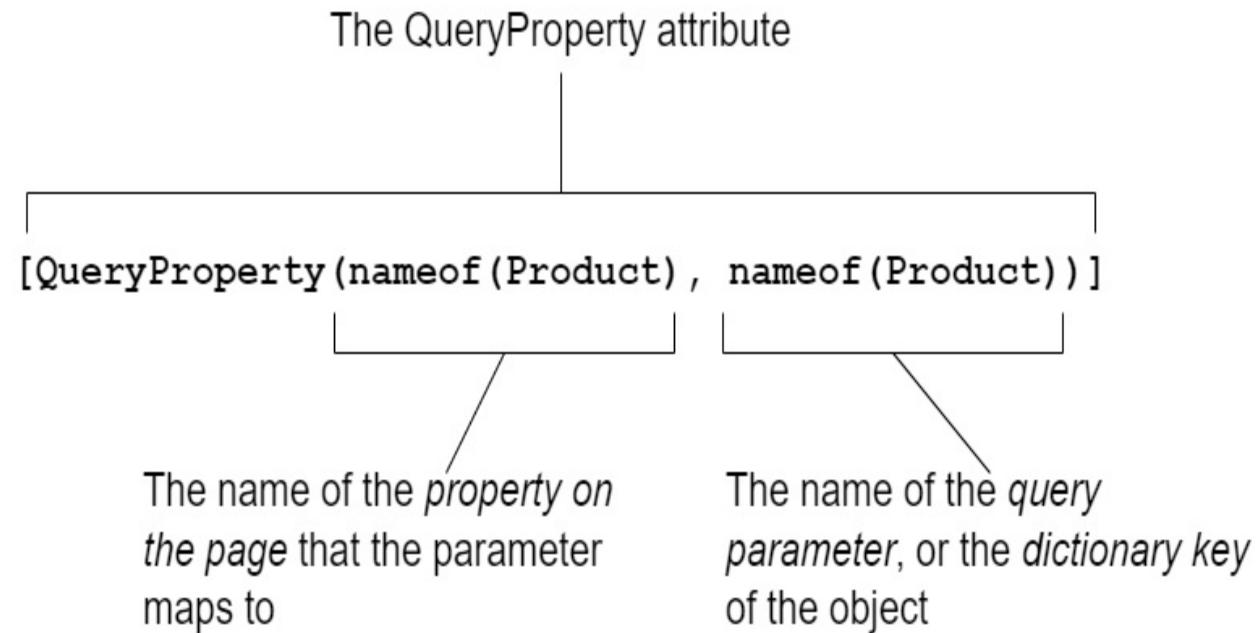
```
private async void Button_Clicked(object sender, EventArgs e)
{
    var product = new Product { Name = "MauiStockTake", Manufac
    var pageParams = new Dictionary<string, object>
    {
        { "Product", product }
    };
}
```

```
        await Shell.Current.GoToAsync("productdetails", pageParams);  
    }
```

You could run the app now, and you'll see the “Go to product” button in the `InputPage`, and if you tap it now, it will take you to the `ProductPage`. But for the `ProductPage` to display any of the data we're passing, we need to set it up to receive query properties.

To do this, we use the `QueryPropertyAttribute` on the page that we want to receive parameters (or on the page's binding context – we'll see more about this in chapter 9). The `QueryPropertyAttribute` has a constructor that takes two arguments: the first is the name of the property on the page that corresponds to the query property, and the second is the name of the query parameter, or the key of the item in the dictionary, that corresponds to the query parameter.

Figure 7.20 The `QueryPropertyAttribute` is used to decorate a page or a page's binding context so that the page can receive URL parameters or navigation state via URL based navigation in Shell



In figure 7.20, we can see how we will use this attribute in the `ProductPage`. If we were using a string, as in the example in figure 7.19, the attribute would

look like this:

```
[QueryProperty("myPagesString", "myvalue")]
```

In 7.16, we passed a query parameter called `myvalue`, so this is what we must bind to here, and this would bind it to a string called `myPagesString` on the page you are navigating to. We could of course use `nameof` here too, to avoid typos. You can add as many `QueryPropertyAttributes` to your pages as you need.

For the `ProductPage`, we'll need to add a `Product` called `Product` to match the key in the navigation state. We're going to display the product name and manufacturer name, so we'll need string properties for these as well as backing fields, and we'll need to call `OnPropertyChanged` in the setter.

Listing 7.17 shows the updated `ProductPage.xaml.cs`

Listing 7.17 ProductPage.xaml.cs updated for navigation state parameters

```
namespace MauiStockTake.Maui.Pages;

[QueryProperty(nameof(Product), nameof(Product))]
public partial class ProductPage : ContentPage
{
    public ProductPage()
    {
        InitializeComponent();
        BindingContext = this;
    }

    Product _product;
    public Product Product
    {
        get { return _product; }
        set
        {
            _product = value;
            ProductName = _product.Name;
            ManufacturerName = _product.ManufacturerName;
        }
    }

    string _productName;
    public string ProductName
```

```

{
get => _productName;
set
{
    _productName = value;
    OnPropertyChanged();
}
}

string _manufacturerName;
public string ManufacturerName
{
get => _manufacturerName;
set
{
    _manufacturerName = value;
    OnPropertyChanged();
}
}
}

```

The last step to display the product is to add Labels and bindings to the XAML, and we can also centre the VerticalStackLayout and add some spacing. Listing 7.18 shows the updated ProductPage.xaml.cs file, with the ContentPage tags omitted. The updated code is in **bold**.

Listing 7.18 The updated ProductPage.xaml

```

<VerticalStackLayout Spacing="50"
    VerticalOptions="Center">
    <Label Text="{Binding ProductName}"
        FontSize="Title"
        VerticalOptions="Center"
        HorizontalOptions="Center" />

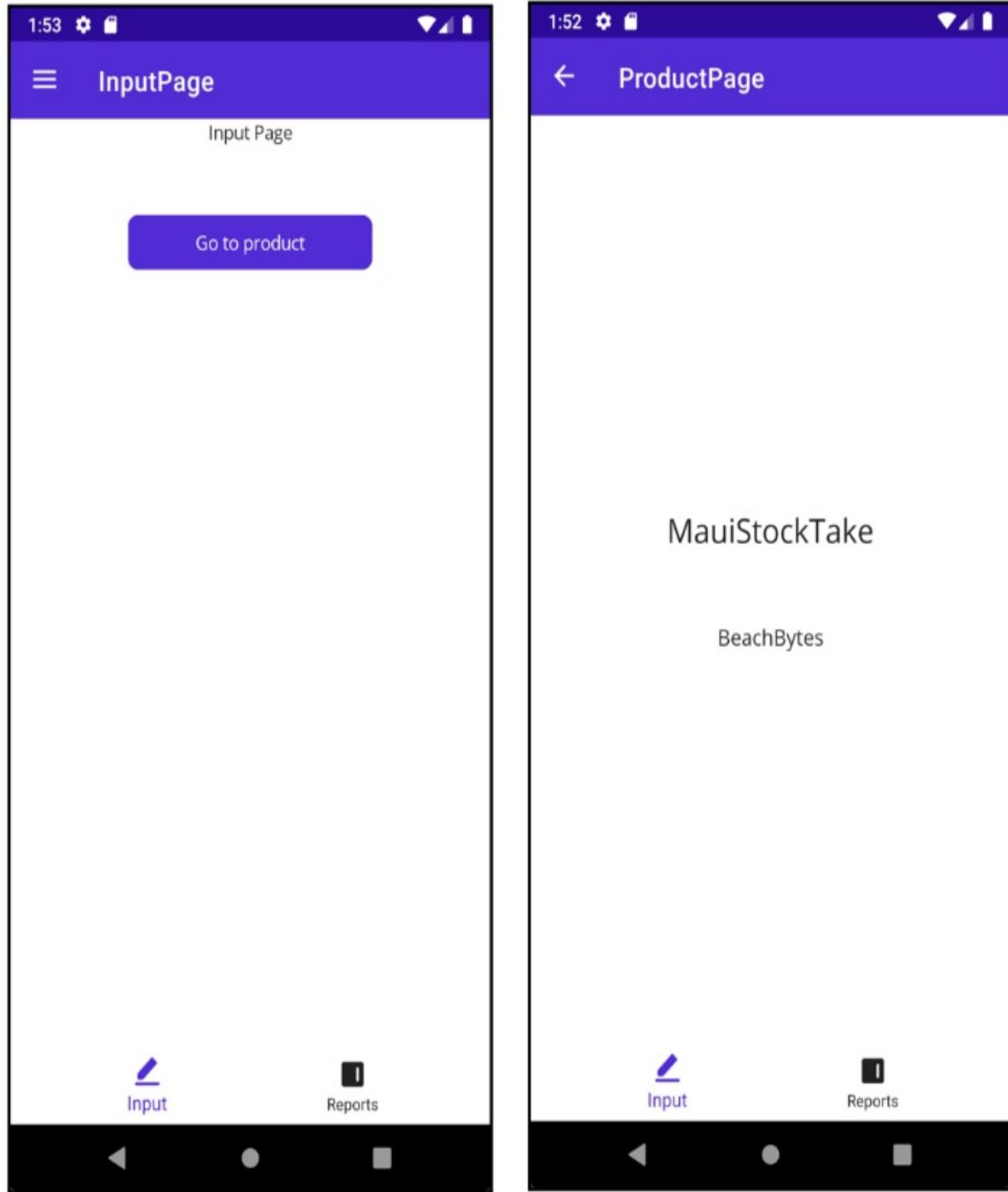
    <Label Text="{Binding ManufacturerName}"
        FontSize="Subtitle"
        VerticalOptions="Center"
        HorizontalOptions="Center" />
</VerticalStackLayout>

```

You can run the app now and should straight away see the “Go to product” button. If you tap it, you should be taken to the ProductPage, and see the details we passed in with the GoToAsync method. You can see both of these in

figure 7.21.

Figure 7.21 In the InputPage on the left, a button is used to programmatically navigate to the ProductPage, using the GoToAsync method. A product is passed in a dictionary, and the ProductPage reads this using the item's key, and assigns properties to strings, which Labels in the XAML are bound to.



We'll revisit the `ProductPage` later in the book, but for now we've got it ready and waiting to accept product details that we can pass in navigation.

7.6 Summary

- You can use `ContentPage` to build out navigable parts of your apps.
- `ContentPage` has a `Content` property, to which you can assign a layout or control. Assign a layout so that you can add multiple controls.
- Common page lifecycle methods allow you to execute code as part of your app's flow, rather than requiring user interaction. You can leverage this to load data when a page appears or perform other actions you might not want your user to have to initiate.
- You can use tabbed navigation, hierarchical navigation, or flyout navigation in .NET MAUI apps. Or you can combine them.
- You can use `Shell` to simplify building an app that has multiple pages. `Shell` lets you define the navigation hierarchy and architecture of the pages in your app in XAML.
- `Shell` provides automatic navigation for flyout items and tabs, but you can navigate programmatically by assigning routes to pages.
- You can pass data around with navigation in `Shell`, using query strings or a dictionary of `<string, object>` to represent navigation state.

8 Enterprise app development

This chapter covers:

- Moving logic out of the UI and into services
- Authentication
- Using the generic host builder pattern to register services and resources
- Dependency injection
- Full stack architecture patterns for .NET MAUI apps

There's a good chance you've chosen to build an app in .NET MAUI because it fits in with your existing .NET stack. Writing apps in .NET MAUI offers several benefits, but perhaps chief among these is the opportunity to leverage your existing skills in .NET and easily integrate a .NET MAUI app into an existing solution or code base.

Many of the patterns and practices you're familiar with from other .NET project types can be used in .NET MAUI apps. In this chapter we'll look at some of these, including abstracting logic into interfaces and implementing that logic in services. We'll also look at some other enterprise patterns in .NET MAUI development, such as dependency injection and authentication, and how we can simplify local development of full-stack, cloud-based solutions with .NET MAUI client apps.

If you've already got your cloud or web infrastructure running on .NET, and you want to add a mobile or desktop app, .NET MAUI is a no-brainer. In this chapter, we'll look at how MauiStockTake.UI slots into our existing .NET API solution, and how we can share code between the different layers of the solution.

8.1 Moving Logic to Services

The MauiStockTake app we began working on in Chapter 7 has a page structure and navigation hierarchy defined in its Shell, but it doesn't do

anything yet. We need to add some functionality for searching for products and adding an inventory count

So far, all the functionality we've been writing in our apps has been in the code behind files for our XAML pages. This is not a smart choice, for a few reasons, but perhaps the two most important are:

- **Tightly couples the UI to business logic:** Writing business logic in the code behind for a Page makes our application more brittle. Changes to the UI can impact the business logic, and vice versa.
- **Doesn't allow code re-use:** Logic written in a Page can't be used by other pages. So if, for example, different pages in the app all need to get information about a product, they each have to implement that logic themselves.

In MauiStockTake we're going to take a different approach and implement the functionality we need in services. This will solve both problems, as removing business logic from the UI breaks that coupling as well as allowing the logic to be used wherever it is needed without having to reimplement it.

Before implementing the services, we'll start by defining requirements. We do this by determining what functionality we will need in the app, and then writing interfaces to describe that functionality.

8.1.1 Defining Requirements

The first thing we want a user to do when they open our app is to log in, so let's start by adding a login `Button` to the `LoginPage`. As well as adding a `Button`, we'll change the layout from a `VerticalStackLayout` to a `FlexLayout`, to better arrange the views, and we'll update the text of the `Label` from "Login Page" to "MauiStockTake". Between the `Label` and the `Button`, we can put the app's logo. We'll use the same `surfshack_logo.jpeg` that we use in the flyout. Finally, we'll wrap the whole layout in a `Grid` and add an `ActivityIndicator` as well, so that we can show the user that something is happening while the login is taking place.

Listing 8.1 shows the updated `LoginPage.xaml` file, with the changes in **bold**.

Listing 8.1 The updated LoginPage.xaml

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiStockTake.UIMauiStockTake.UI.Pages.Logi
    Title="LoginPage">
    <Grid>
        <FlexLayout JustifyContent="SpaceAround"
            Direction="Column">#A
            AlignItems="Center">#B
            HorizontalOptions="Center"
            VerticalOptions="Center">
            <Image Source="surfshack_logo.jpeg"
                WidthRequest="200"
                HeightRequest="200"
                HorizontalOptions="Center"
                VerticalOptions="Center">
                <Image.Clip>
                    <EllipseGeometry Center="100,100"
                        RadiusX="100"
                        RadiusY="100"/>
                </Image.Clip>
            </Image>
            <Label Text="MauiStockTake"
                FontSize="Title"
                VerticalOptions="Center"
                HorizontalOptions="Center" />
            <Button Text="Login"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="LoginButton_Clicked"
                x:Name="LoginButton"/>
        </FlexLayout>
        <ActivityIndicator x:Name="LoggingIn"
            IsRunning="True"
            IsVisible="false"/>
    </Grid>
</ContentPage>
```

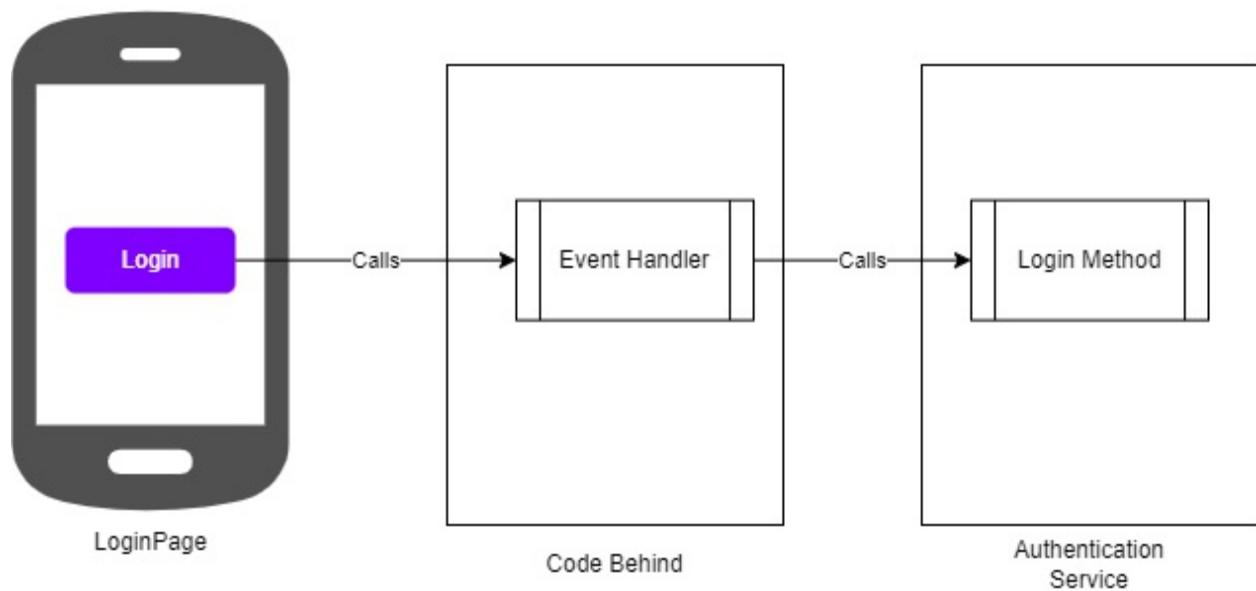
You can see that the login button has a `Clicked` event handler defined, so let's go ahead and add this. Listing 8.2 shows the method to add to the `LoginPage.xaml.cs` file.

Listing 8.2 The LoginButton_Clicked method

```
private void LoginButton_Clicked(object sender, EventArgs e)
{
}
```

We now have our first *requirement* defined. We have a login button, so we have a requirement for functionality to let the user log in. When we use the dependency inversion principle (see MVVM for SOLID Apps in the next chapter), we define requirements where they are consumed (rather than where they are provided), so we can start by defining an interface to describe the functionality we need to consume on the LoginPage. This requirement is summarized in figure 8.1.

Figure 8.1 The LoginPage has a Login button. This triggers an event handler in the code behind file. The event handler doesn't handle the process of logging in itself, instead it calls a Login method in an authentication service.



In the `MauiStockTake.UIMauiStockTake.UI` project, create a new folder called Services, and in here create an interface called `IAuthService.cs`. We know that the implementation of this interface is going to let the user log in, so we can add a method called `LoginAsync`. We'll need to know whether the login was successful or not, so we should have it return a `bool` (we could handle this a bit more elegantly, but this will suit our needs). Finally, the login process will communicate with the REST API, so we want this to

happen on a background thread so the UI doesn't freeze while we're waiting for a response, so we should make the return type a Task.

Listing 8.3 shows the code for the `IAuthService` interface.

Listing 8.3 The `IAuthService` interface

```
namespace MauiStockTake.Maui.Services;
public interface IAuthService
{
    Task<bool> LoginAsync();
}
```

We've now defined a namespace that we're going to use in a lot of places throughout our app, so we should add this to our `GlobalUsings.cs` file. Add the `MauiStockTake.UIMauiStockTake.UI.Services` namespace to `GlobalUsings.cs`:

```
global using MauiStockTake.UIMauiStockTake.UI.Services;
```

We can't quite run the app yet as there are two problems for us to solve. The first is that there is no implementation of this interface. This won't stop us running the app, but it will throw a `NullReferenceException` if you click the `Login` button. The second is that, while we've registered a route for the `LoginPage`, we haven't given the user any way to navigate to it.

We'll provide an implementation for the `IAuthService` in the next section, but for now we can create a mock implementation. In the `Services` folder, create a new class that implements the `IAuthService` interface called `MockAuthService`. For the implementation of the `Login` method, simply return a Task result of `true`. Listing 8.4 shows the code for the `MockAuthService`.

Listing 8.4 `MockAuthService`

```
namespace MauiStockTake.UIMauiStockTake.UI.Services;
public class MockAuthService : IAuthService
{
    public Task<bool> LoginAsync() => Task.FromResult(true);
}
```

Mocking interfaces

We've added the mock implementation for `IAuthService` so that we won't be held up by not having a real implementation. Defining the requirements in interfaces lets you get unblocked while you're building a UI but don't have the functionality you depend on yet, as you can provide a mock implementation in the meantime.

This won't help you do anything that depends on your user actually being logged in (like authenticating API calls for example), but it will allow you to build and run your app and test your UI, and it can be useful for unit testing later as well.

Now that we've got a mock implementation for the `IAuthService` interface, we can add some functionality to the `LoginButton_Clicked` event handler. In the `LoginPage.xaml.cs` file, add a field for the `IAuthService`, and in the constructor assign a new instance of the `MockAuthService` to it. In the event handler, the first thing we'll do is disable the login button, to prevent the user from tapping it while a login is in progress (this won't be an issue for our mock implementation but it's a good practice). Then we can call the `Login` method in the authentication service, and if it was successful, return the user into the main guts of the app, and if not, display a warning. Of course, we know that our mock implementation is just going to return `true`, but this will set us up well for when we provide a real implementation. Listing 8.5 shows the full code for the `LoginPage.xaml.cs` file.

Listing 8.5 LoginPage.xaml.cs

```
namespace MauiStockTake.UIMauiStockTake.UI.Pages;

public partial class LoginPage : ContentPage
{
    private readonly IAuthService _authService; #A

    public LoginPage()
    {
        InitializeComponent();
        _authService = new MockAuthService(); #B
    }
}
```

```

private async void LoginButton_Clicked(object sender, EventArgs e
{
    LoginButton.IsEnabled = false;#C
    LoggingIn.isVisible = true;#D

    var loggedIn = await _authService.LoginAsync();#E

    LoggingIn.isVisible = false;

        if (!loggedIn)#F
    {
        await App.Current.MainPage.DisplayAlert("Error", "Something went
            LoginButton.IsEnabled = true;
            LoggingIn.isVisible = false;
        }
    else
    {
        // TODO: navigate back to the app
    }
}
}

```

We've got a `TODO` comment in the code for when the login was successful. That's because at the moment we're not actually showing the `LoginPage`, so there's nothing to go back from. Logging in is the first thing the user will need to do, so we should show it automatically when the app starts. We've got a route registered for `LoginPage`, but we should remove this route as we want the `LoginPage` to appear as part of the app's lifecycle, rather than being navigable (you can read more about application lifecycle here: <https://docs.microsoft.com/dotnet/maui/fundamentals/app-lifecycle>).

We'll use the `OnStart` application lifecycle method and use hierarchical navigation to push the `LoginPage` as a modal page onto the navigation stack. Unlike a `NavigationPage`, a modal page doesn't provide navigation UI (i.e. a back button), which gives the user a cue that the actions in the page should be completed before navigating away from the page, which is well suited for our login scenario (of course on Android you have a hardware or OS supplied back button, so the user can still dismiss the page, but the cue is still there, and you can override this functionality if necessary).

In the `App.xaml.cs` file, override the `OnStart` method, and within the method push a modal instance of the `LoginPage` onto the stack. The navigation stack

is managed by a property called `Navigation` that is inherited by the `Page` class from the `NavigableElement` base class. The app's navigation stack is passed by reference to the page currently assigned to the app's `MainPage` property, so we can use this path to call the navigation methods. Listing 8.6 shows the code to add to `App.xaml.cs`.

Listing 8.6 The OnStart functionality to add to App.xaml.cs

```
protected override async void OnStart()
{
    base.OnStart();

    await MainPage.Navigation.PushModalAsync(new LoginPage()); #A
}
```

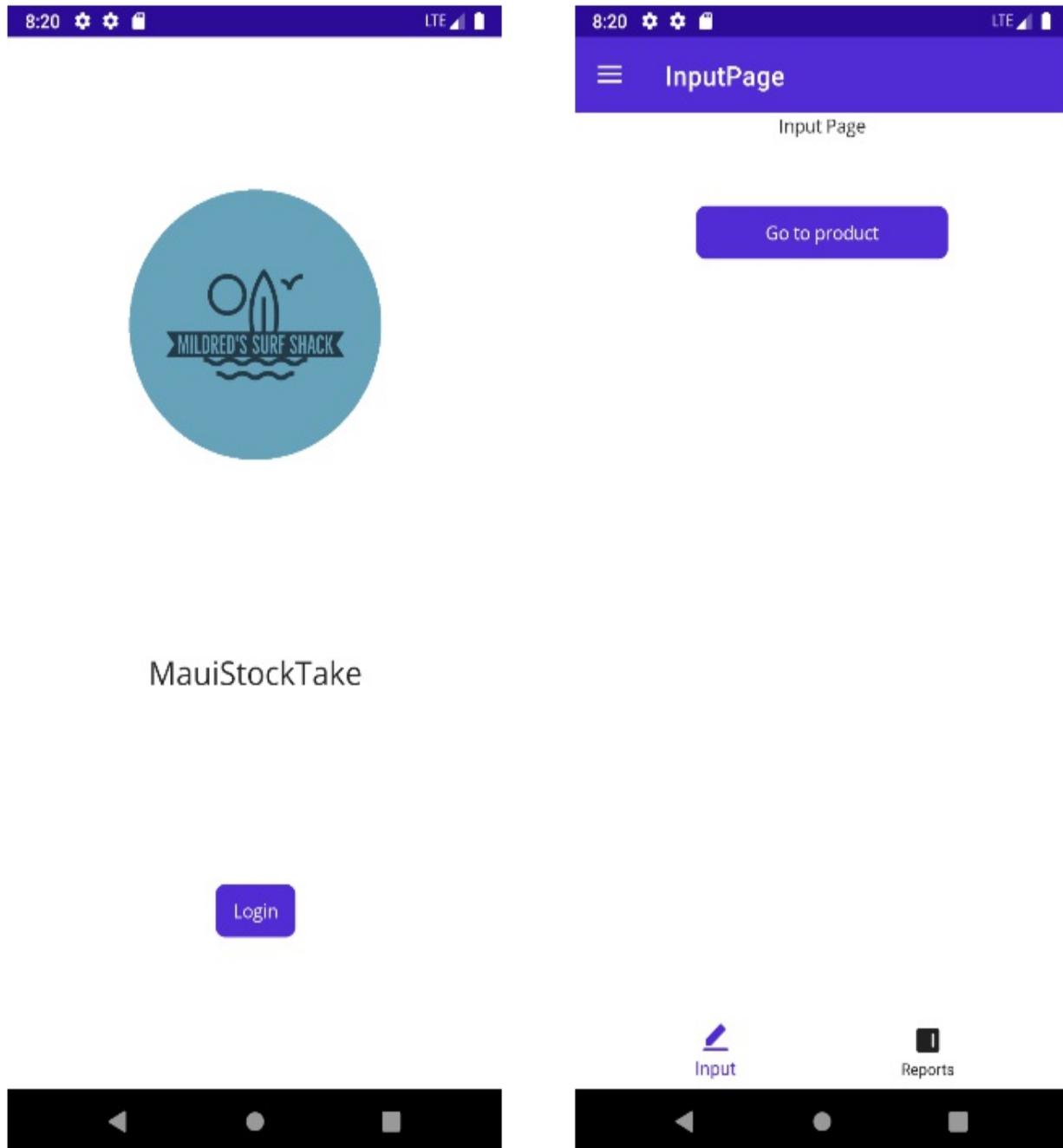
This will push a modal instance of the `LoginPage` when the app starts. Once the user has logged in, we can pop this page to return to the main Shell. `LoginPage` also inherits `NavigableElement`, so it also has a `Navigation` property which holds a reference to the app's navigation stack. This means we can call the `PopModalAsync` method on `LoginPage`'s `Navigation` property. Replace the `TODO` comment in `LoginPage.xaml.cs` with the code in listing 8.7.

Listing 8.7 The code to add to LoginPage.xaml.cs in place of the TODO

```
await Navigation.PopModalAsync();
```

Run the app now and you should see the `LoginPage` pop up automatically. Click the login button and the page will disappear, and you'll be back to the app Shell. You can see the result in figure 8.2.

Figure 8.2 On the left, the LoginPage has been pushed as a modal page onto the navigation stack automatically when the app starts. Note the app does not provide a back button. If you click on the Login button, the LoginPage is popped programmatically, revealing the existing Shell underneath.



8.1.2 Implementing the Authentication Service

The MauiStockTake API uses IdentityServer, an OpenID Connect (OIDC) compliant framework for ASP.NET Core applications. OIDC, an extension of OAuth2, lets users authenticate using their web browser to obtain a token that can be used to access protected resources. We'll need to build an

authentication service in the .NET MAUI app that can sign our users in using IdentityServer and obtain a JSON web token (JWT) that can be used to authenticate calls to the API.

Identity, authentication, and authorization

Identity is a huge topic and way outside the scope of this book, but it's an important topic that you should at least have some familiarity with. There are plenty of apps that don't require authentication or have any user accounts, but these are a small minority. If you'd like to learn more about identity and authentication, you can check out my video here:

<https://youtu.be/z0LBNdIjhpg>.

You don't have to use IdentityServer in your apps, in fact it's become increasingly popular to outsource identity to a cloud-based identity provider (IDP), like Azure Active Directory B2C or Auth0. We're using IdentityServer here because it's included in the ASP.NET Core templates, so it doesn't require signing up to a third-party service, and you may already be familiar with it. Also, most cloud based IDPs offer a well-documented and supported client package or SDK that you can install and use in your apps and are very easy to use. They often also take care of some of the functionality we're going to build ourselves, like securely storing refresh tokens.

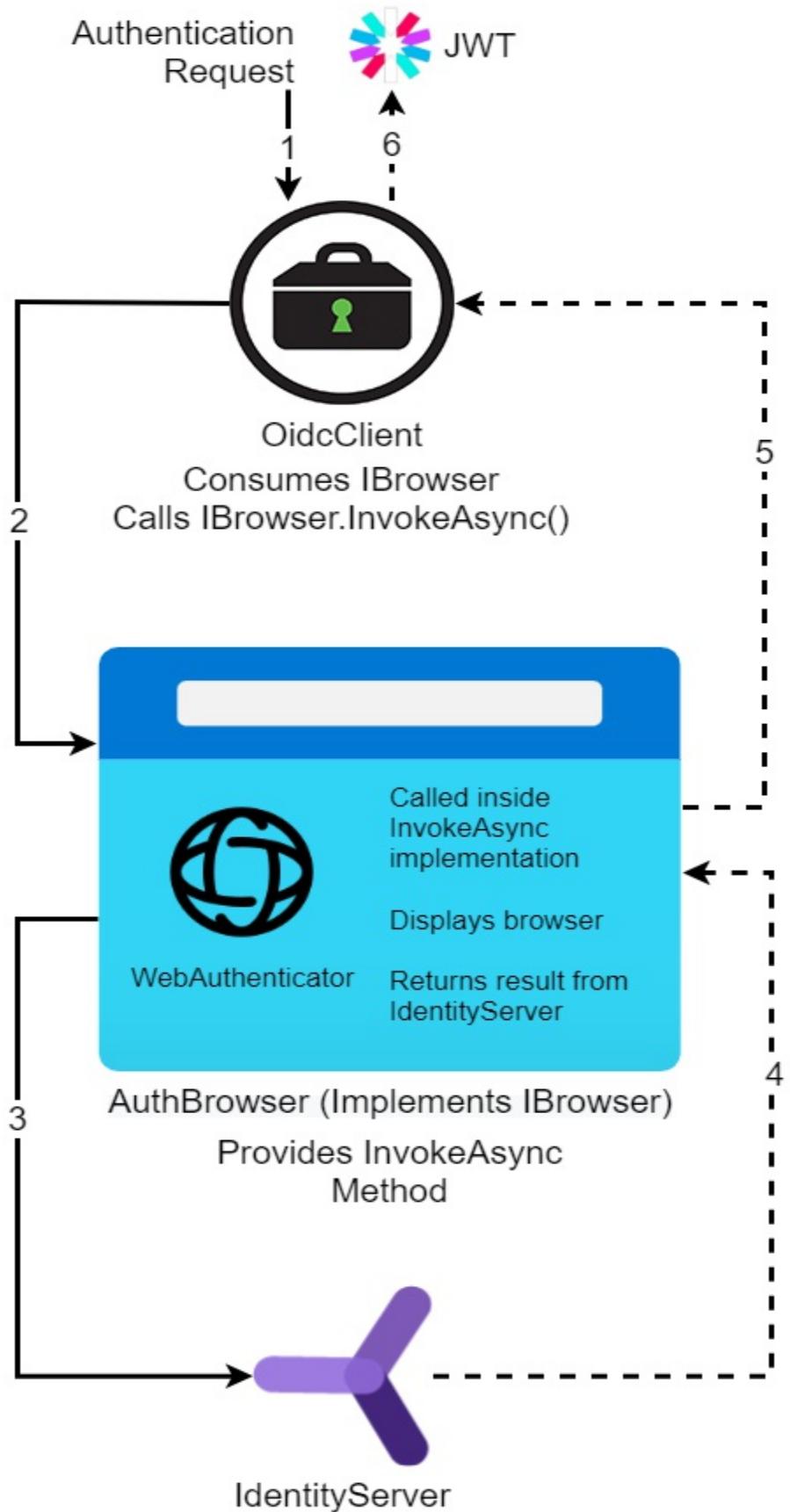
If you're outsourcing identity in your apps, it will be a lot easier to get started with documented client packages, but if you want to use IdentityServer, the approach here will help. And it will also work with any OIDC compliant IDP, including most of the commercial cloud offerings.

With OAuth2 authentication, rather than sending a username and password from your app to the IDP, your app opens a web browser at the IDP's login page. This is considered more secure, as your app never has access to the user's password; they login to the IDP directly, and once they're logged in, they get redirected back to the app with a code that can be used to obtain an access token.

To achieve this in our `IAuthService` implementation, we'll use a combination of the built-in `WebAuthenticator` that comes with .NET MAUI

and a NuGet package called IdentityModel.OidcClient. OidcClient is made by the same people that make IdentityServer, and simplifies the process of parsing the OAuth2 response that IdentityServer returns for a logged in user. Figure 8.3 shows how this will work in MauiStockTake.

Figure 8.3 1. Call the `LoginAsync` method of the `OidcClient` package. 2. The `OidcClient` package will call `WebAuthenticator` and display the login page on `IdentityServer`. 3. The user will log in to `IdentityServer`. 4. `IdentityServer` returns the result via `WebAuthenticator`. 5. The parsed result is returned to `OidcClient`. 6. `OidcClient` extracts the JWT from the result and returns it to the caller of the `LoginAsync` method.



`OidcClient` is used to parse an OIDC response but doesn't provide a way to direct the user to an IDP's login page to authenticate. When creating an `OidcClient` instance, you need to provide an implementation of their `IBrowser` interface, which defines a method for performing this action. We'll build an implementation of this interface that uses `WebAuthenticator` to perform the login, then pass the results back to `OidcClient`, which will extract the tokens we need and return them to our authentication service.

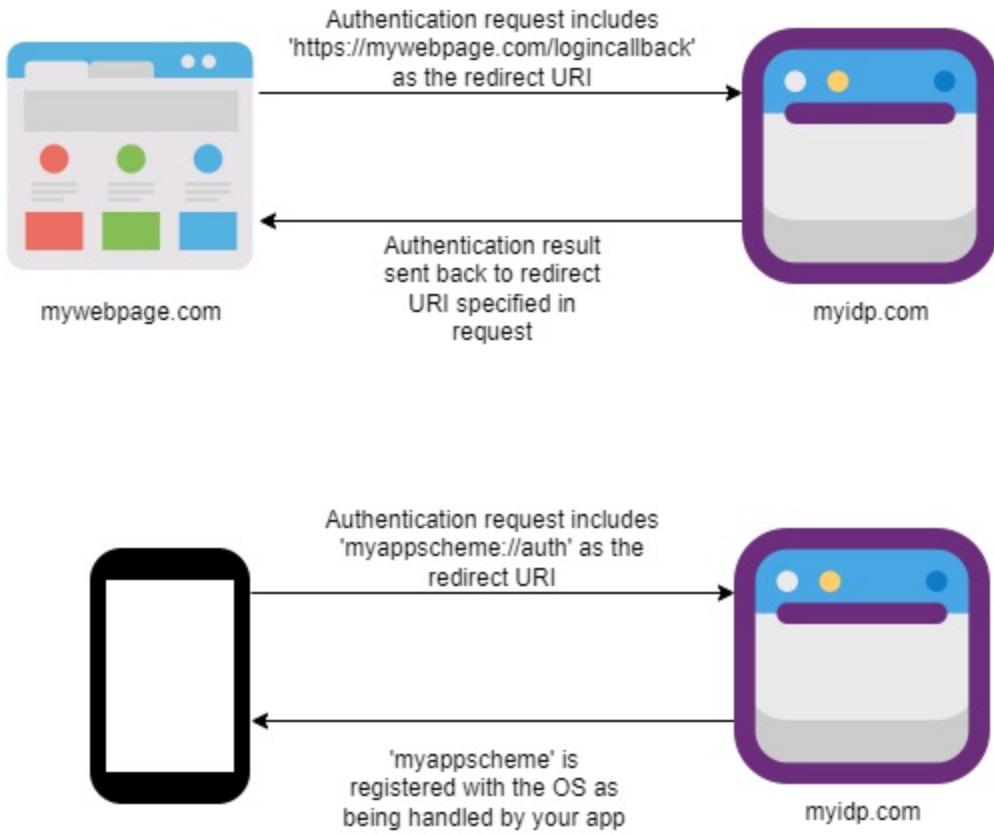
NOTE

We'll need to use the fully qualified name for `IBrowser`, as .NET MAUI also has an interface called `IBrowser`, with a different use case.

When we use an OAuth2 login flow with a web browser, as part of the request we provide a redirect URI. Once the user successfully authenticates, the IDP returns them to the redirect URI, along with their access token and a refresh token. In our case, we don't want to redirect the user to a website, but to our app. To do this, we register a custom URL scheme with the operating system, so that the OS knows that any URLs bound to addresses starting with that scheme should be sent to our app.

No doubt you are familiar with the `http://` and `https://` URL schemes. When you open a URL, it will be opened with the default application registered with the operating system for handling the scheme. For HTTP and HTTPS, this will be your default web browser. By using a custom scheme, we can associate it with our app, so any URL beginning with that scheme will be opened in our app. Figure 8.4 shows how we can utilise this to get the authentication response back from an IDP.

Figure 8.4 When authenticating with an IDP, a redirect URL is supplied. Upon successful authentication, a response is sent back to the redirect URL. For a web application, this is usually the address of the web application where the authentication request originated. For a mobile or desktop app, the URL uses a custom scheme that the OS associated with the app, so the response from the IDP is sent to the app.



For our application, we'll use `auth.com.mildredsurf.stocktake://callback` as the redirect URI. In this case, `auth.com.mildredsurf.stocktake` is the scheme, so we need to register this scheme with our target platforms. The process is slightly different for each platform, and the following sections will walk through setting this up for each OS that our app can target.

Android URL Registration

Registering a custom URL scheme for Android requires two steps. First, we need to create an Activity that will receive the web callback. Second, we need to add an intent to the manifest that will allow us to open the browser.

Let's start with the Activity. Add a file in Platforms/Android called `WebCallbackActivity.cs`, and add the content from listing 8.8.

Listing 8.8 WebCallbackActivity.cs

```
using Android.App;
using Android.Content;
using Android.Content.PM;

namespace MauiStockTake.UIMauiStockTake.UI.Platforms.Android;

[Activity(NoHistory = true, LaunchMode = LaunchMode.SingleTop, E
[IntentFilter(new[] { Intent.ActionView },
    Categories = new[] { Intent.CategoryDefault, Intent.CategoryB
    DataScheme = "auth.com.mildredsurf.stocktake",
    DataHost = "callback")]
public class WebCallbackActivity : WebAuthenticatorCallbackActivi
{

}
```

Next, we will tell Android that our app will use custom tabs. Custom tabs sit between an embedded webview and switching apps to another browser (you can find out more about them here:

<https://developer.chrome.com/docs/android/custom-tabs>) Add the bold code from listing 8.9 to the `AndroidManifest.xml` file (as we saw in chapter 3, can be found in the `Platforms/Android` folder), before the closing `</manifest>` tag.

Listing 8.9 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/andro
<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="30
<application android:allowBackup="true" android:icon="@mipmap/ap
<uses-permission android:name="android.permission.ACCESS_NETWORK_
<queries>
<intent>
<action android:name="android.support.customtabs.action.CustomTa
</intent>
</queries>
</manifest>
```

iOS and Mac Custom URL Registration

The process for registering a custom URL scheme with iOS and macOS is the

same. Repeat the following steps in the platform folders for iOS and Mac Catalyst.

First, register the custom URL scheme in the `info.plist` file (remember from chapter 3 this can be found in the Platforms/iOS and Platforms/MacCatalyst folders). If using Visual Studio, double click on the file to open the file with the plist editor, and go to the Advanced tab. Expand the URL Types node, and click on Add URL type. Add the following values:

Table 8.1 The values to add to info.plist

Field	Value
Identifier	Auth
URL Schemes	auth.com.mildredsurf.stocktake
Role	Viewer

If you are not using Visual Studio, or you would simply prefer to add these values manually, open `info.plist` with a text editor. Inside the `<dict>...</dict>` tags, add a new entry to the dictionary with a key of `CFBundleURLTypes`. The value will be an array, which will also contain a dictionary. Listing 8.10 shows you how to add the entries for the custom URL scheme registration. Add these before the closing `</dict>` tag.

Listing 8.10 Custom URL definition in Info.plist for iOS and macOS

```
<key>CFBundleURLTypes</key>
<array>
<dict>
<key>CFBundleURLName</key>
<string>Auth</string>
<key>CFBundleURLSchemes</key>
```

```
<array>
<string>auth.com.mildredsurf.stocktake</string>
</array>
<key>CFBundleTypeRole</key>
<string>Viewer</string>
</dict>
</array>
```

Don't forget to repeat these steps in both the iOS and MacCatalyst platform folders.

Windows Custom URL Scheme Registration

To register the custom URL scheme on Windows we need to update the Package.appxmanifest file. Open the file in a text editor (if using Visual Studio, right click and select View Code, or click on the file in Solution Explorer and press F7). Inside the manifest, locate the Applications node and the Application node with the Id of App inside it.

Add the code from listing 8.11 directly after the <Application ...> opening tag.

Listing 8.11 Custom URL definition in Package.appxmanifest for Windows

```
<Extensions>
<uap:Extension Category="windows.protocol">
<uap:Protocol Name="auth.com.mildredsurf.stocktake">
<uap:DisplayName>Auth</uap:DisplayName>
</uap:Protocol>
</uap:Extension>
</Extensions>
```

Now that we've registered the URL scheme on each platform, we'll need some helper classes to handle the response from IdentityServer.

Defining the IdentityServer values

When interacting with an OAuth2 IDP, we need a few values defined to help us locate the IDP and specify some details needed for the login interaction. Specifically, these are the URI of the IDP, the client ID (which represents a

client defined in the IDP, in this case our MauiStockTake app), the scopes we request access to, and the redirect URI.

We'll make all these values constant and the class static so that they are easy to access from anywhere in the app.

Add a new class in the root of the MauiStockTake.UIMauiStockTake.UI project called `Constants`. This class will contain several values that we'll use throughout the app. Listing 8.12 shows the code for the `Constants` class. The code here shows ngrok URLs I have generated for testing (see Appendix A). Replace these with your own ngrok URLs (see appendix A) when you want to run the app (or whatever URLs you are using for testing).

Listing 8.12 Constants.cs

```
namespace MauiStockTake.Client;

public static class Constants
{
    public const string BaseUrl = " https://8284-159-196-124-207.

    public const string RedirectUri = "auth.com.mildredsurf.stoc

    public const string AuthorityUri = " https://8284-159-196-124

    public const string ClientId = "com.mildredsurf.stocktake";#D

    public const string Scope = "openid profile offline_access M
}
```

These values will let you connect to a demo instance of the MauiStockTake API. If you want to host this yourself, or run it locally and connect using ngrok, update the values for the `BaseUrl` and `AuthorityUri` accordingly.

Add the AuthBrowser

The `AuthBrowser` class will implement the `IBrowser` interface defined in the `OidcClient` NuGet package, so we need to install this first. **Install IdentityModel.OidcClient into the MauiStockTake.UI project.** Next, in

the MauiStockTake.UI project, add a folder called `Helpers`, and in here add a class called `AuthBrowser`.

We'll need to declare that `AuthBrowser` implements the `IBrowser` interface, but as .NET MAUI also has an `IBrowser` interface, we'll need to use the fully-qualified name including the namespace. `IBrowser` defines a method called `InvokeAsync`, so we will need to implement this method.

This method returns a type called `BrowserResult`. It has a single string property called `Response`, which provides the values returned by an IDP formatted in a way that `OidcClient` can parse. So, in the `InvokeAsync` method, we'll invoke `WebAuthenticator` which will provide the values from the IDP in a dictionary called `Properties`. We'll write another method to read these and add the values that `OidcClient` expects to a string formatted in a way it can read, then add this string the `Response` property of a `BrowserResult` and return this to the caller.

Listing 8.13 shows the code for the `AuthBrowser` class.

Listing 8.13 the AuthBrowser class

```
using IdentityModel.OidcClient.Browser;
namespace MauiStockTake.Client.Helpers;

public class AuthBrowser : IdentityModel.OidcClient.Browser.IBrowser
{
    public async Task< BrowserResult> InvokeAsync(BrowserOptions
    {
        WebAuthenticatorResult authResult = await WebAuthentica

        return new BrowserResult() #D
        {
            Response = ParseAuthenticationResult(authResult) #E
        };
    }

    private string ParseAuthenticationResult(WebAuthenticatorResu
    {
        string code = result?.Properties["code"];
        string scope = result?.Properties["scope"];
        string state = result?.Properties["state"];
    }
}
```

```

        string sessionState = result?.Properties["session_state"]
        return $"{Constants.RedirectUri}#code={code}&scope={scope}"
    }
}

```

This completes the `AuthBrowser` class, which gives us a method of opening a web browser at the IDP login page and return a formatted result that `OidcClient` can process. That's all the preparation we need and can now build the `AuthService`.

Add the AuthService

In the Services folder, create a class called `AuthService`. The `AuthService` will implement the `IAuthService` interface which, at the moment, defines a single method called `LoginAsync`.

In the `AuthService`, we're going to implement the `LoginAsync` method using the `OidcClient`. When we instantiate this client, it will require some options in its constructor, and we'll build these using a combination of values we set in the `Constants` class, and the `IBrowser` implementation we created.

The `OidcClient` will use `WebAuthenticator` to invoke a browser session and capture the returned session state. The key part of this we need to authenticate against the API is the access token. We're going to see how we can use this access token to make authenticated calls to the API shortly, but for now we just want to check the result to see whether there has been any error. If not, we know we've authenticated successfully and have an access token, and can return `true` from this method, otherwise return `false`.

Listing 8.14 shows the code for the `AuthService` class.

Listing 8.14 The AuthService class

```

using IdentityModel.OidcClient;
using MauiStockTake.UI.Helpers;

namespace MauiStockTake.UI.Services;

public class AuthService : IAuthService#A
{

```

```

private readonly OidcClientOptions _options;#B

public AuthService()
{
    _options = new OidcClientOptions#C
    {
        Authority    = Constants.AuthorityUri,
        ClientId    = Constants.ClientId,
        Scope        = Constants.Scope,
        RedirectUri = Constants.RedirectUri,
        Browser      = new AuthBrowser()
    };
}

public async Task<bool> LoginAsync()#D
{
    var oidcClient = new OidcClient(_options); #E

    var loginResult = await oidcClient.LoginAsync(new LoginRe
        if (loginResult.IsError)#G
    {
        // TODO: inspect and handle error
        return false;
    }

    return true;
}
}

```

Now that we've got a real implementation of the `IAuthService`, we can go back to the `LoginPage` code behind and update the constructor. Change the line that assigns the `MockAuthService` to the `IAuthService` field to use the real `AuthService` instead.

```
_authService = new AuthService();
```

If you've followed the setup instructions in Appendix A, you can now run the API and use ngrok (or tunnel of your preference) to get a publicly routable URL. Ensure this URL is in your `Constants` file, both for the authority and base URL.

Run the app now, and it should automatically push a modal instance of the login page. If you click the login button, on iOS and Android you'll see a

browser window open within the app, and on Windows and macOS your default browser will open to the login page of IdentityServer. The API automatically creates a default account, with a username of `administrator@localhost` and a password of `Administrator1!`; you can log in with these credentials or register an account for yourself.

Once you log in, if you are on iOS or Android the browser window will disappear, or if you’re on macOS or Windows the browser will ask for permission to open the MauiStockTake app (make sure you approve this). You’ll then be back in the app, and the login page will be popped from the stack, just like with the `MockAuthService`.

If you want, you can put a breakpoint in the `AuthService` on the line that checks the `loginResult` to see if it is an error. You can inspect `loginResult`, and you should see an access token, an ID token and a refresh token. Now that our app has a way to obtain an access token, we can build the functionality to make authenticated API calls and make the app do something useful.

8.2 Using the Generic Host builder and Dependency Injection

.NET MAUI uses the same generic host builder pattern that’s used in other .NET project types, such as console or ASP.NET Core applications. This means it includes a built-in dependency injection (DI) container.

NOTE

If you’re not already familiar and comfortable with DI, you can read up on it in *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann, Manning 2019.

We can use this in our .NET MAUI apps too. In the `LoginPage` code behind and in the `AuthService`, we are “newing up” a bunch of dependencies for the login process that we should be able to resolve from the DI container (the `IBrowser` implementation in the `AuthService`, and the `AuthService` in the `LoginPage`).

Before we can consume any services or dependencies from the DI container, we need to register them, and in order to register them, we need to decide their scope. With an ASP.NET Core application, you have a clearly defined HTTP request pipeline, and each request has a scope. This can often inform the service scope you use to register dependencies.

With a UI app, things work slightly differently. As there's no HTTP request pipeline, registering something with `AddScoped` doesn't make any sense, because there aren't requests to scope for. This leaves two remaining options: transient or singleton.

With a Shell app in .NET MAUI, pages added to the Shell via XAML or registered for routing (for example, the `InputPage`, `ReportPage` and `ProductPage`) are automatically registered in DI with a singleton scope, but for pages outside of Shell (for example, our `LoginPage`), and for other dependencies like services and ViewModels (which we'll learn about in the next chapter), we need to register them manually.

With that in mind, we'll register our dependencies according to the rules in table 8.2:

Table 8.2 Scope lifetimes for different dependencies in a .NET MAUI app

Dependency type	Scope	Reasoning
Pages	Transient	We should expect a new instance of a page every time we request one. Persisting values across different instances of a page could cause us problems, so we're better off persisting state elsewhere and using transient instances of pages.
ViewModels	Transient	Just like pages, our ViewModels should be transient, so we know we've got a clean instance

		every time we get a new ViewModel.
Services	Singleton	We should only expect one instance of a service for the running lifetime of an app. Instantiating multiple copies of a database, or a service responsible for communicating with an API, is a waste of resources and can lead to data conflicts. These should be singletons and can therefore be where we persist app-wide state. We can inject these singleton instances into ViewModels.

Why are we scoping pages differently to Shell?

I mentioned above that Shell automatically registers pages as singleton but then suggest registering pages manually as transient. This is really a question of UX and user expectations.

In a Shell app, a user expects to be able to navigate between pages quickly and usually expects to see the same instance of a page as they navigate back and forth. So the most efficient use of resources is to register the page as singleton and persist it in memory for the lifetime of the app.

With other navigation paradigms, such as hierarchical (and windowed, which we'll see in chapter 10), having a single instance of a page or its state can be problematic, as the user could expect a fresh instance every time, and persisting seldom used items in memory throughout the lifetime of the application is inefficient.

Using the Facebook app as an example, the home tab displays the newsfeed, and if it reloaded every time you tab away and then back to it, you would consider this poor UX. Within the news feed, if you tap on a profile, it isn't navigable within the tabs, but rather pushes hierarchically onto the stack. A fresh instance of the profile page makes sense every time, otherwise it will hold stale data from previous profiles that you have visited.

Using the generic host builder pattern, we can register pages, ViewModels

and services as we would in an ASP.NET Core or console app. But with .NET MAUI there are other things we can register too. Over the next few sections, we'll look at each of these.

8.2.1 Registering Resources, Services and Other Dependencies

We already have some experience with registering resources. In chapter 4 we added an LCD font for the MauiCalc app, and in chapter 6 we registered an icon font for the Outlook replica.

In the `MauiProgram` class, we can see that the `CreateMauiApp` method returns a `MauiApp`. This method is called by the framework to create an instance of the app for each target platform. Inside this method, the generic host builder pattern is used to generate the `MauiApp` that gets returned.

Using this pattern, an extension method called `ConfigureFonts` allows us to register the fonts we want to use in our app. If you've previously used `Xamarin.Forms`, you'll appreciate how much simpler this is with .NET MAUI. Other extension methods are used for registering other things, like handlers and animations (we'll learn about both in chapter 11).

We don't need to register any fonts in `MauiStockTake`, but we do have some dependencies that need to be registered.

The host builder in `MauiProgram` is a temporary variable called `builder` of type `MauiApplicationBuilder`. It has a property called `Services` of type `IServiceCollection`, and this is what we'll use to register dependencies, and what the framework will use to resolve them for us.

Let's work backwards. Looking at the `AuthBrowser`, we can see that it has no external dependencies. It implements an interface defined by `oidcClient` and uses the `WebAuthenticator` provided by .NET MAUI. So we can start by registering this interface implementation with the service collection.

In `MauiProgram.cs`, after the `ConfigureFonts` call and before the builder is returned from the method, register the `AuthBrowser` as an implementation of `IBrowser` with singleton scope:

```
builder.Services.AddSingleton<IBrowser, AuthBrowser>();
```

You will need to bring in the required namespaces, but if you add this line as-is now, you will get an error. This is because if the duplicate `IBrowser` name in .NET MAUI and `OidcClient`. To solve this, you can use the fully qualified name, or to make it easier to read, declare at the top of the file that `IBrowser` in this case means the `OidcClient` version:

```
using IBrowser = IdentityModel.OidcClient.Browser.IBrowser;
```

The next link in the chain is the `AuthService`. `AuthService` implements the `IAuthService` interface, so we can register this implementation as a singleton as well. Add this after the `IBrowser` registration:

```
builder.Services.AddSingleton<IAuthService, AuthService>();
```

Most of the pages will be managed by `Shell`, but the `LoginPage` is a modal navigation page so we need to register it ourselves. After the `AuthService`, register the `LoginPage` with transient scope:

```
builder.Services.AddTransient<LoginPage>();
```

Listing 8.15 shows the full code for the updated `MauiProgram` class.

Listing 8.15 MauiProgram.cs with the dependency registrations

```
using MauiStockTake.UI.Helpers;
using MauiStockTake.UI.Pages;
using IBrowser = IdentityModel.OidcClient.Browser.IBrowser;

namespace MauiStockTake.UI;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRe
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansS
```

```

        });

        builder.Services.AddSingleton<IBrowser, AuthBrowser>();

        builder.Services.AddSingleton<IAuthService, AuthService>()

        builder.Services.AddTransient<LoginPage>();

        return builder.Build();
    }
}

```

That's all the dependencies we have in MauiStockTake at the moment. Now that they're registered, let's see how we can consume them.

8.2.2 Consuming Services

We can consume dependencies from the service collection in .NET MAUI using constructor injection, just like in any other .NET application.

AuthService has a dependency on IBrowser, but we can only see this by reading the code, as it is not currently constructor injected. But now that it's registered with the service collection, we can constructor inject it and assign the injected instance to the _options field.

In the AuthService, change the constructor to take an injected instance of IBrowser, and assign this to the Browser property of the _options field. You will also need to add the same using statement as we added to MauiProgram to resolve ambiguity between the two IBrowser interfaces.

Listing 8.16 shows the updated constructor for the AuthService.

Listing 8.16 The updated AuthService constructor

```

using IBrowser = IdentityModel.OidcClient.Browser.IBrowser;
...
public AuthService(IBrowser browser)
{
    _options = new OidcClientOptions
    {
        Authority    = Constants.AuthorityUri,
        ClientId    = Constants.ClientId,
    }
}

```

```

        Scope      = Constants.Scope,
        RedirectUri = Constants.RedirectUri,
        Browser     = browser
    };
}

```

Now that we've changed the constructor of the `AuthService`, if you open the `LoginPage` you will see that we have an error in the constructor. We are currently assigning a new `AuthService()` instance to the `_authService` field, but `AuthService` doesn't have a default constructor anymore. `AuthService` now has a visible dependency on `IBrowser`. We could change this to `_authService = new AuthService(new AuthBrowser);`, but it's better to just inject the `IAuthService` into the `LoginPage`. This way the service collection will be responsible for providing the fully resolved `IAuthService` implementation and we adhere to the dependency inversion principle by depending on the requirements defined by the `LoginPage`, rather than on any specific implementation. Listing 8.17 shows the updated constructor for the `LoginPage`.

Listing 8.17 The updated LoginPage constructor

```

public LoginPage(IAuthService authService)
{
    InitializeComponent();
    _authService = authService;
}

```

Now that we've changed the constructor for `LoginPage`, we have an error in `App.xaml.cs`, in the `OnStart` method. We are passing a new instance of `LoginPage` to the `PushModalAsync` navigation method, but, again, we are depending on a default constructor which no longer exists.

There are a few ways we could solve this issue. For example, we could add a static property of type `IServiceCollection` to `MauiProgram` and assign the `Services` property of the `MauiApplicationBuilder` to it. Then we could call this from anywhere in our app to resolve dependencies; but this would be using the *service locator* anti-pattern.

Alternatively, we could inject `LoginPage` into the `App` class, assign it to a field, and then pass this field to the `PushModalAsync` method.

There is a better approach though, which is to use the `PageResolver` NuGet package (disclaimer: I am the author of this package). Using this package, you can navigate to pages by type, using the page as a type argument. The plugin will then navigate to a fully resolved instance of the page with all its dependencies.

Install the `Goldie.MauiPlugins.PageResolver` NuGet package into the `MauiStockTake.UI`. We'll need to refer to this package in a couple of places, so let's add it to the `GlobalUsings` file:

```
global using Maui.Plugins.PageResolver;
```

Now we can update the code in `App.xaml.cs` to use the simplified navigation method and pass the `LoginPage` as a type parameter without having to worry about its dependencies:

```
await MainPage.Navigation.PushModalAsync<LoginPage>();
```

The last step is to register the `PageResolver` with the generic host builder, which will pass it the service collection which it can use to resolve dependencies. In `MauiProgram.cs`, append `UsePageResolver()` to the fluent `UseMauiApp` method. Listing 8.18 shows the updated method.

Listing 8.18 the updated `UseMauiApp` method

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

Everything is now in place to automatically resolve and consume all of `LoginPage`'s dependencies. Run the app now, and you should be able to log in just as you could before (you still need your API and tunnel running and may need to update your tunnel URL).

There's no change in functionality here from what we had before, but the

code is now more maintainable.

8.3 Consuming Web Services

Many apps are self-sufficient and don't have any external dependencies, but enterprise apps, and most successful consumer apps, usually need to communicate with an API. Many technologies like REST, SignalR, GraphQL and gRPC are available to facilitate this connectivity. We're using REST in MauiStockTake as it's still the most prevalent, but we'll look at using gRPC in .NET MAUI apps in chapter 11, and you can see a sample of a chat app using SignalR here: <https://github.com/matt-goldman/maui-chat>.

When using REST, the API represents resources and clients interact with them using the HTTP verbs (GET, POST, PUT, PATCH and DELETE). Payloads are sent in JSON format, which allows APIs and clients to communicate without concern for what technology each implements behind the scenes.

In a .NET application, you can use an extension method in the `System.Net.Http.Json` namespace to call a REST endpoint using `HttpClient`, and deserialize the JSON response to a .NET type:

```
var product = await _httpClient.GetFromJsonAsync<Product>("produc
```

In this example, `_httpClient` is an instance of `HttpClient` that has its `BaseUrl` defined (which would make "product" a route).

This is a good approach and we've used it a couple of times already (in the Outlook replica and in MauiMovies). For larger applications this can become difficult to maintain, but in MauiStockTake we are using Clean Architecture (CA), which uses NSwag to automatically generate clients for each resource.

Auto-generated clients

Nswag is a convenient tool for generating client class libraries for .NET APIs. For a trivial API like MauiStockTake it's probably not necessary, but I am using it as it's built into the CA template. As your solutions grow in complexity and the number of routes in the API increases, the value of an

auto-generated client increases exponentially.

If you prefer not to use it, you can write your own methods to interact with the API, based on the specification. You can explore the API specification by running the WebAPI project and appending /api to the URL. This will bring up the Swagger UI.

You are free to choose whichever approach you wish, but I recommend using the auto-generated client. This will make it easier for you to follow along with the code in this chapter, and will save you time, allowing you to focus on .NET MAUI specific topics, rather than writing REST clients.

The REST client implementations are already taken care of in the MauiStockTake solution. In this section, we'll see how we can wire these up in our .NET MAUI app.

8.3.1 Adding the Client Project

The MauiStockTake solution has a class library project called MauiStockTake.Client. If you look in the `Helpers` folder, you'll see an auto-generated file that contains client classes for interacting with the REST endpoints. In the `Services` folder, you'll find services for the Product and Inventory resources, along with interfaces that you can inject where you need them, that provide a usable wrapper for these clients. We'll look at these in a bit more detail shortly.

We need to add the MauiStockTake.Client project as a dependency on MauiStockTake.UI. If you're using Visual Studio, right-click on **Dependencies** under MauiStockTake.UI, and click **Add Project Reference...**. From there, you can check the box next to MauiStockTake.Client and click Ok. If you open the MauiStockTake.UI.csproj file, you'll see that this adds the following lines:

```
<ItemGroup>
    <ProjectReference Include="..\MauiStockTake.Client\MauiStockTake.Client.csproj" />
</ItemGroup>
```

If you're not using Visual Studio, you can add these manually to add the

project reference.

MauiStockTake.Client has a dependency on another project in the solution called Shared which contains DTOs used by the API and the client project. One of these is called `ProductDto`, and this is almost identical to the `Product` class we created in the `Models` folder of MauiStockTake.UI. We don't need this class anymore, so you can delete the `Product` class and the `Models` folder.

This will cause errors in `ProductPage.xaml.cs` and `InputPage.xaml.cs`, so let's fix these. First, in each page remove the `using` statement for the `Models` namespace and replace it with a `using` statement for `MauiStockTake.Shared.Products`. Next, in both page code behind files, update every mention of `Product` to `ProductDto`. Build the MauiStockTake.UI project to ensure there are no errors.

8.3.2 Using a Delegating Handler

The MauiStockTake API expects requests to include an access token sent as a header, and requests made to API routes that require this token that do not include it will not be authorized. It's straightforward to add this header to any `HttpRequestMessage`, but it can become a lot to handle if we have to do this every time we call the API.

Instead, we can use a delegating handler. A delegating handler can be associated with an `HttpClient` instance and can modify every HTTP request made by that client so that you don't have to do it yourself on every call. Simply call an HTTP method using that client, and the modification will be applied to the request automatically.

In the MauiStockTake.Client project, there's already a delegating handler set up. Look in the `Authentication` folder of the MauiStockTake.Client project and you'll see a class called `AuthHandler`. This class inherits the `DelegatingHandler` base class and overrides the `SendAsync` method. Inside this method the logic is simple – add a header to the request that includes the access token, and then just call the base method.

Notice that this class also has two additional members. The first is a static

string that will hold the value of the access token. The overridden `SendAsync` method uses this to attach to the request, but at the moment this value is not being populated.

Looking back at our `AuthService` in the `MauiStockTake.UI` project, we can see that in the `LoginAsync` method, when we get a successful result we're simply returning true. Let's update this to set the value of the access token to the static string in the `AuthHandler`. As it's a static string, we can refer to it using the class name and member name without needing an instance.

In the `AuthService`, in the `LoginAsync` method, add a line before we return true to assign the value of the access token from the `loginResult` to the `AuthToken` member of the `AuthHandler`. You will also need to bring in the namespace for the `AuthHandler`. Listing 8.19 shows the updated `LoginAsync` method, with the added code in **bold**.

Listing 8.19 The updated LoginAsync method of AuthService

```
using MauiStockTake.Client.Authentication;
...
public async Task<bool> LoginAsync()
{
    var oidcClient = new OidcClient(_options);

    var loginResult = await oidcClient.LoginAsync(new LoginReques
        if (loginResult.IsError)
    {
        // TODO: inspect and handle error
        return false;
    }

    AuthHandler.AuthToken = loginResult.AccessToken;

    return true;
}
```

The delegating handler is now ready to modify any HTTP request to include the access token. Now we need a way for our API client classes to get access to an `HttpClient` instance that has this handler attached.

8.3.3 Using IHttpClientFactory

Microsoft provides a NuGet package called `Microsoft.Extensions.Http` that enables the use of `HttpClient` with dependency injection (this package is already installed in the `MauiStockTake.Client` project). Using an extension method in this package you can call `AddHttpClient` to add a singleton instance of `HttpClient` to the services collection.

You can call this method multiple times to add multiple instances of `HttpClient`, each one serving a different purpose and each referenced with a unique name (you can also register `HttpClient` instances for specific types). When you require an instance of `HttpClient`, you inject the `IHttpClientFactory` interface into your class, and call its `CreateClient` method to resolve the named instance of the client you require from the services collection.

Recall that in the `AuthHandler` we had a const string called `AUTHENTICATED_CLIENT`. The purpose of this string is to provide a name that we can refer to throughout the app for the instance of `HttpClient` that will add the token to HTTP requests.

In the root of the `MauiStockTake.Client` project is a file called `DependencyInjection` which contains a method called `AddApiClientServices`. Note in this method that we are adding the delegating handler to the services collection:

```
services.AddSingleton<AuthHandler>();
```

Immediately following this, an `HttpClient` instance is registered, named using the const from the `AuthHandler` class, and with the `AuthHandler` added to its handler chain:

```
services.AddHttpClient(AuthHandler.AUTHENTICATED_CLIENT)
    .AddHttpMessageHandler((s) => s.GetService<AuthHandle
```

This registers an instance of `HttpClient` with the services collection, that can be consumed anywhere in the app, that has a handler attached that will append the access token to any HTTP request made using this client.

Chaining delegating handlers

You can do as much as you like to manipulate the `HttpRequestMessage` in a delegating handler before you call the base method to send it. We only have one modification (attaching the access token), but there may be circumstances where you need more. For example, one product I have worked on requires specific headers depending on certain criteria.

You can make all these changes in one delegating handler, but it's better to create specific handlers for each modification. You can call `AddHttpMessageHandler` multiple times to chain handlers.

This approach better adheres to the single responsibility principle, but also gives you more flexibility. Handlers will be processed in the order that they are added, so with this approach you can also control the order in which these modifications are applied.

In the `MauiStockTake.Client` project, look in the `Services` folder. In it you can see three files:

- `BaseService`
- `InventoryService`
- `ProductService`

`BaseService` contains a base class that expects `IHttpClientFactory` to be injected. The base service contains a field for an instance of `HttpClient`, and the constructor requests an instance of the named client, using the name defined in the `AuthHandler`, and assigns it to this field:

```
public BaseService(IHttpClientFactory httpClientFactory, ApiClientOptions options)
{
    _httpClient = httpClientFactory.CreateClient("AuthHandler");
    _baseUrl = options.BaseUrl;
}
```

You can see here that the constructor also expects a class called `ApiClientOptions` and uses that to populate a base URL field; we'll talk about this in the next section.

As all services will use the same token, the `InventoryService` and `ProductService` inherit this base class. Their constructors must also take the dependencies that the base class requires and pass them through to the base constructor, but they don't need to double-handle requesting the `HttpClient` instance from `IHttpClientFactory`.

Looking at the constructor of either the `InventoryService` or `ProductService` class, you can see that they use this authenticated client to create an instance of the REST resource-specific client used by each of these services. Using a managed `HttpClient` instance gives us an access token to make authenticated requests to any endpoint in the API, without having to handle it on each individual request.

8.3.4 Adding the remaining MauiStockTake services

In the `MauiStockTake.UI` project, we've already seen how we can wire up services and dependencies in the `IServiceCollection`. In `MauiProgram`, we've registered `AuthService` as an implementation of the `IAuthService` interface.

The `MauiStockTake.Client` project also has some interfaces defined as well as some implementations (we looked at their constructors in the previous section). These dependencies are already registered in the `DependencyInjection` class, so all we need to do to use them in the `MauiStockTake.UI` project is add this existing dependency registration.

The `AddApiClientServices` method is an extension method on `IServiceCollection`, so we can register it in `MauiProgram`. Registration of the service interfaces and their implementations is already done in this method, so adding this in `MauiProgram` will make these services available in the app.

As it's an extension method, we can call it on an existing `IServiceCollection`, rather than passing the `IServiceCollection` in. But the method also expects another parameter of type `ApiClientOptions`. In `MauiProgram`, call this extension method on the `Service` property of the `builder` variable, and pass in a new instance of `ApiClientOptions`, with

your ngrok (or preferred tunnel) URL as the `BaseUrl`. Listing 8.20 shows the code to add to `MauiProgram`.

Listing 8.20 Registering the client in MauiProgram

```
builder.Services.AddApiClientServices(new ApiClientOptions
{
    BaseUrl = "https://75e1-159-196-124-207.au.ngrok.io"
});
```

With the `MauiStockTake.Client` project now registered, we can inject the service interfaces into our classes and use them to talk to the API. Authentication is handled by the `AuthHandler`, and the services are wired up to use an instance of `HttpClient` that uses this handler.

In the next chapter, we'll finish building the stock-taking functionality of this app by using these services.

8.4 Full-stack App Architecture

`MauiStockTake` is not a standalone mobile and desktop application. It's part of a larger solution that includes cloud/web components and a database, and it requires authentication to secure communication between the .NET MAUI app and the cloud API.

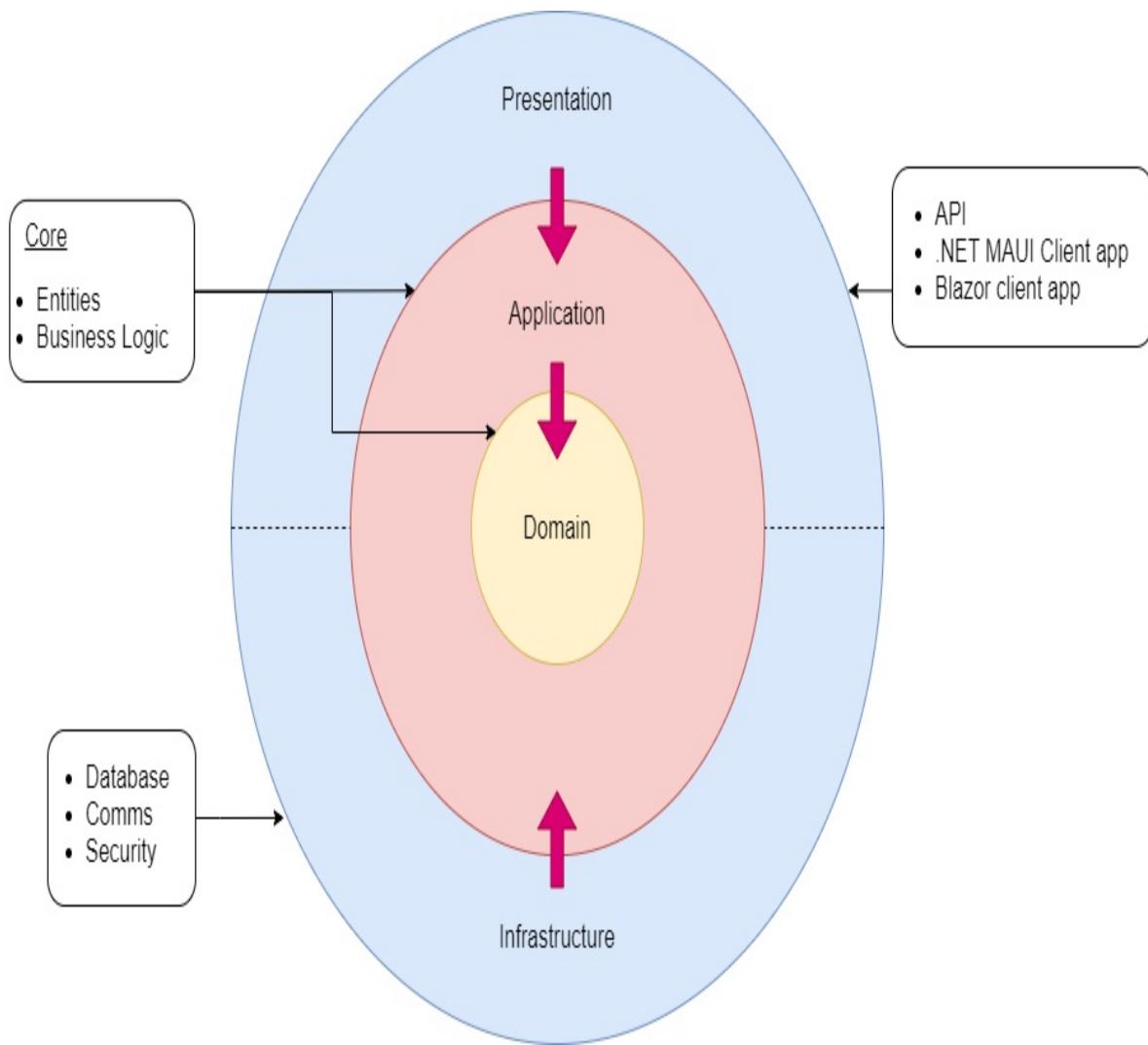
In this section, we'll look at how to organize your solution to maximize code sharing and efficiency.

8.4.1 Project Organization

As mentioned in the last chapter, the `MauiStockTake` API is based on the Clean Architecture (CA) template and follows clean architecture principles, but the code sharing techniques we've covered in this chapter will let you integrate a .NET MAUI UI into any full-stack .NET architecture. Figure 8.5 shows how the API projects in the solution are organized.

Figure 8.5 With Clean Architecture, all dependencies point inwards. In the Core (Domain and Application) are your entities and business logic. Application depends on Domain, and Domain

has no dependencies. Infrastructure and Presentation depend on Application, and the .NET MAUI app is part of Presentation.



The Clean Architecture pattern, as applied to the back-end API, is beyond the scope of this book, but you can learn more about it by watching Jason Taylor's talk from NDC Sydney here: <https://www.youtube.com/watch?v=5OtUm1BLmG0>. The particulars of the architecture are not that important; what's important is that we're following the principles and a design pattern to maximize code re-use across our whole stack.

You don't have to use Clean Architecture

I used CA for MauiStockTake because it provides a logical, structured way to organize code in your solution. As I use it every day at work, and often help

Jason to teach it, this familiarity makes it a lower cognitive burden for me to use CA when starting a new project, but more importantly, this structure helps to illustrate how we can share code between the API and the .NET MAUI app.

There are plenty of alternative architectures though; some may be better suited to your scenario, or you may simply not like the CA approach (many people don't). And that's fine. Micro-services and event-driven architectures in particular are currently in vogue.

Whether you use CA or some other architecture, the principles of sharing code across your stack are the same, and the techniques used here will work just as well with other architectures.

As part of this solution, it makes sense to re-use code by sharing it among the different projects where possible, as we saw in section 8.3. For now, let's focus on the Presentation layer, as this is where our focus will be in building a UI.

The solution is arranged in folders representing the different layers of CA, and looking in the `Presentation` folder, we see:

- `WebAPI`
- `MauiStockTake.Maui` (we added this in the last chapter)
- `Client`

WebAPI

The `WebAPI` project is an ASP.NET Core project that provides REST controllers and endpoints that allow the outside world to communicate with the business logic and data in the API. The .NET MAUI app communicates with this over HTTP to interact with the rest of the solution.

MauiStockTake.UI

This is the .NET MAUI project that we added in the previous chapter. This is where we will build the app that Mildred and her team will use when taking

inventory.

MauiStockTake.Client

`MauiStockTake.Client` is a class library that contains types and logic that can be used in a .NET UI project for interacting with the API. It contains DTOs and services, as well as some auto-generated client code.

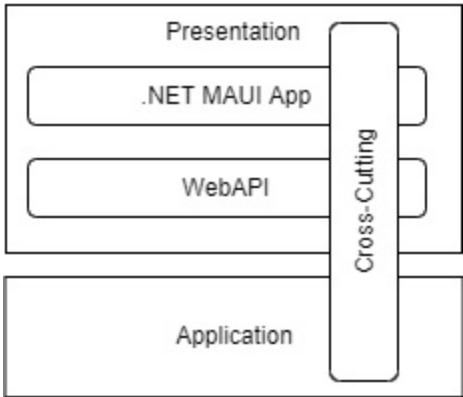
Everything in this project is essential for making the `MauiStockTake.UI` client app work, but it's boilerplate code that is not part of the .NET MAUI app itself. In fact, if we wanted to add a Blazor web app in the future, the Blazor app can reuse this class library, eliminating the need to duplicate code that achieves the same thing.

8.4.2 Sharing Code between Projects in the solution

In Clean Architecture, it's a strict rule that dependencies point inwards. It's also a strict rule that the Domain has no external dependencies. Typically, Application is part of Core along with Domain, and Application *can* have dependencies; just not dependencies that point outwards.

Parts of the solution that run perpendicular to the flow of dependencies are referred to as *cross-cutting concerns* (see Figure 8.6). In the case of `MauiStockTake`, we have cross-cutting concerns in the form of DTOs that are required by the Application project, the WebAPI project, and the .NET MAUI app (although they are provided to the .NET MAUI app via the API client project).

Figure 8.6 `MauiStockTake` is a full-stack solution written in C#. Therefore, there is no need to duplicate code as it can be shared across the stack. Code shared across the stack in this way is referred to as cross-cutting concerns.



In MauiStockTake, these DTOs are in a project called Shared (in a solution folder called Common). This Shared project is a dependency for the Application project, the WebAPI project, and the MauiStockTake.Client project. Using this approach, we can share code between the backend and front end effortlessly.

Sharing code in this way provides some significant advantages. It adheres to the DRY principle, which doesn't just make life easier, it gives us assurances that changes in one part of the solution will be reflected instantly elsewhere. If we change the structure of a DTO for example, that same DTO is already in use everywhere, so any breaking changes are instantly recognizable. This is much more difficult when the layers of the stack are developed independently.

But as we're building a full-stack .NET solution, we can take this a step further. In the future, if we decide to add a web UI to the solution, we can use Blazor, and the MauiStockTake.Client package can be used in the Blazor UI too. This gives us a full-stack cloud, mobile, desktop, and web solution that maximises code reuse across the whole stack.

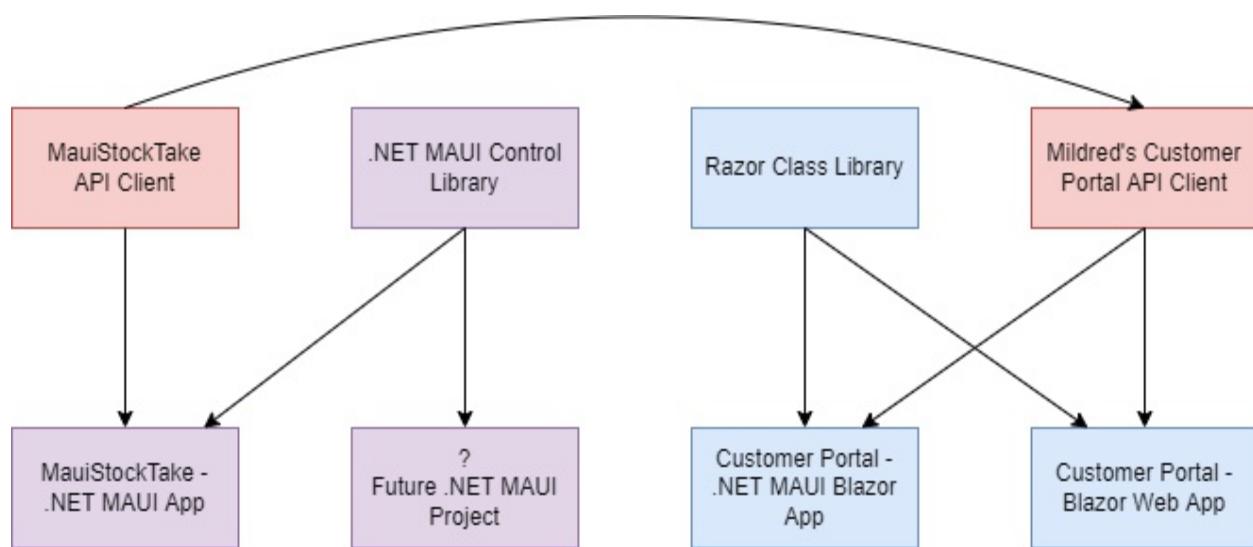
8.4.3 Sharing Code between Solutions

As a standalone solution, MauiStockTake is already well architected for maximum code reuse. Often, though, the software we build forms part of an enterprise ecosystem. Enterprise logic often forms the core of these ecosystems, providing logic and types that are relevant across the whole enterprise and in this way distinct from domain-problem-specific types and questions.

For example, imagine a suite of applications in an enterprise, each one helping with a specific line of business. They would all have their own requirements but would likely share some common functionality. The most obvious example would be user management and authentication.

In addition to business logic, it's important for these applications to maintain a consistent UX to make them feel like part of a cohesive whole, especially with externally facing products. .NET MAUI makes it easy to do this. Figure 8.7 shows an example of how the MauiStockTake app could form part of an enterprise ecosystem.

Figure 8.7 The MauiStockTake app becomes part of an ecosystem if Mildred introduces a customer portal as well, which may require (among other things) an indication of availability of products. The customer portal would likely have its own API, which could query the MauiStockTake API for this information. The customer portal could be built in Blazor, which would allow its controls to be abstracted back to a Razor class library, which could be used by .NET MAUI Blazor to provide the customer portal as an app too. The controls in the MauiStockTake app could be abstracted back to a .NET MAUI class library, to be shared with future apps too, to maintain the visual consistency of Mildred's brand.



In this scenario, Mildred's enterprise has a suite of applications, both consumer facing and internal, that can not only share business logic (where applicable), but UI and UX as well. In figure 8.7, the whole ecosystem is broken down into modules that can be shared across solutions. A common approach is to bundle these as NuGet packages and host them on a private feed. GitHub offers NuGet package hosting (public and private), and many other products exist that can integrate with your preferred DevOps or CI/CD

platform.

At this point, we've reached the nirvana of code sharing in a .NET enterprise ecosystem. We've got business logic and UI that's shared across the whole enterprise, in the cloud, on the desktop, and in the browser.

A full-scale enterprise solution like this is beyond the scope of this book, but you can see a demonstration of this approach in my GitHub repo here: <https://github.com/matt-goldman/CloudyMobile>.

8.5 Summary

- Moving logic out of UI and into services lets you share it across a solution.
- You can define requirements for your UI by creating an interface. You can then write a service that implements this interface. This is an example of the dependency inversion principle.
- The WebAuthenticator in .NET MAUI makes it easy to authenticate using OAuth in your apps. It opens the browser session for you and returns the token response to your code.
- You can register a custom URL scheme with the OS for your app. This allows a web browser (or any other application) to route to your app using a URL.
- .NET MAUI uses the generic host builder pattern used across all .NET application types.
- Using the host builder, you can register fonts for use in your .NET MAUI apps.
- The generic host builder pattern gives you access to the built-in services collection. You can use this to register dependencies and inject them into your classes via their constructors.
- You can create a delegating handler which can be configured to automatically attach a header with an access token to any HTTP request made by an `HttpClient`.
- You can register a named instance of `HttpClient`, along with a delegating handler, in the services collection. You can inject `IHttpClientFactory` into any class and use it to get access to the named `HttpClient` instance.

- With .NET MAUI, sharing code across your whole stack is easy. You can place DTOs in a shared project that the frontend and backend both have access to.
- You can even share business logic - in MauiStockTake, the client package could be used by a Blazor UI too.
- Sharing logic and UI across an enterprise is also easy with ASP.NET Core, .NET MAUI and Blazor, using familiar CI/CD and DevOps tools.

9 The MVVM pattern

In this chapter:

- The Model, View, ViewModel paradigm
- The problems that MVVM solves in complex apps
- Making Controls MVVM friendly with Behaviors

The Model, View, ViewModel (MVVM) pattern was introduced by Microsoft with WPF and has become the standard for apps developed using XAML. It's a popular pattern with enough nuance that entire books have been dedicated to the subject. This chapter will provide an introduction, and as you progress through your .NET MAUI journey, you may find that this is a topic that you want to explore in more depth.

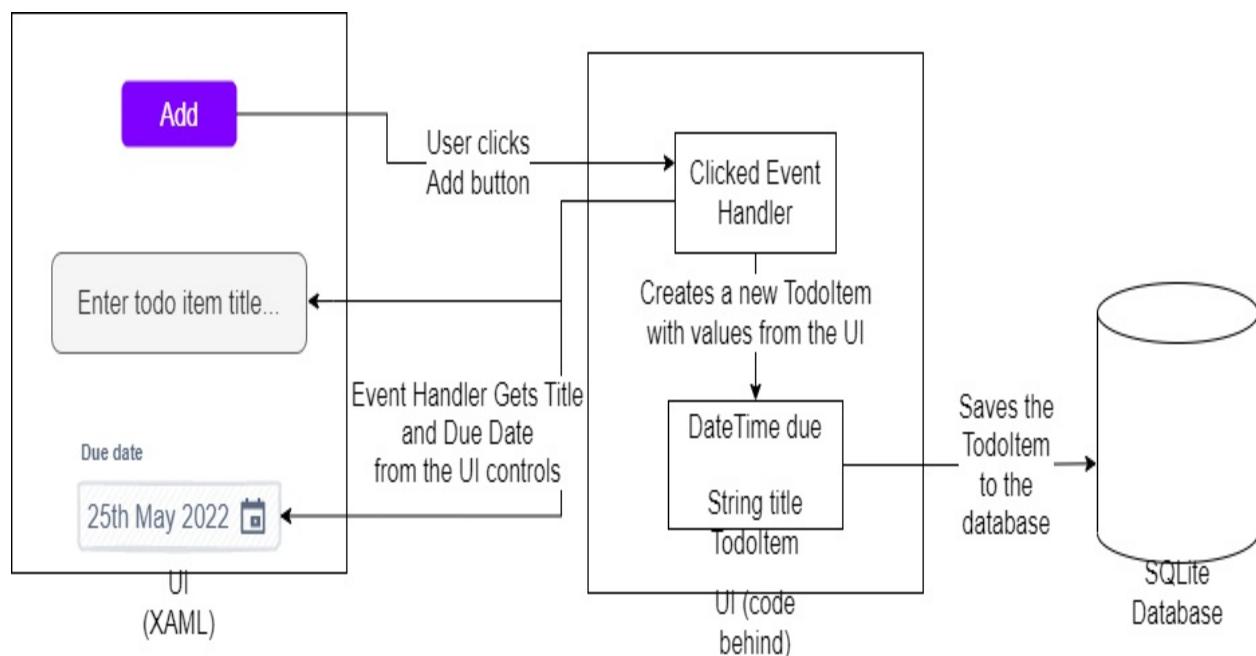
Use of the MVVM pattern is, in some ways, more an art than a science. You should aim to understand the rules about separating UI logic, presentational logic and business logic if you are adopting the MVVM pattern in your app. Of course, blind adherence to any pattern is an anti-pattern, and you should prioritise ensuring that your code is readable and maintainable. But you can't make an informed decision about when to deviate from the pattern if you don't understand it well.

In this chapter, we'll see how by adopting the MVVM pattern we can make our apps simpler to build and maintain. Before we get into the details of the principles of the pattern, we'll see how in practice it can solve real-world problems. Specifically, to help understand the pattern, we'll refactor MauiTodo to solve our outstanding problem of not marking to-do items as done. Once we've seen the practical advantages, we'll dive deeper into the philosophy of the pattern, and then refactor MauiStockTake for MVVM and use the MVVM pattern to complete the stock taking feature.

9.1 Refactoring the MauiTodo app for MVVM

In chapter 3 we saw how data binding can be used to improve our to-do app, by binding to a collection and using a template to display each item in the collection. Let's look at how we can take this a step further and use the MVVM pattern to solve one significant outstanding issue with MauiTodo. Figure 9.1 shows the current data architecture of MauiTodo, without MVVM.

Figure 9.1 Without the MVVM pattern, the UI for the MauiTodoApp handles everything, including communicating with the database. When a user clicks the Add button, the event handler retrieves the new item title and new item due date, and creates a new to-do item with these values, then saves it to the database. The UI cannot currently save changes to the checked state of an item to the database.



An important feature of a to-do app is the ability to mark items as complete. MauiTodo has a **CheckBox** in the data template that renders alongside every to-do item, but it doesn't do anything.

The problem is that while we can add an event handler to the code behind to respond to **CheckBox** events, we have no way of passing in any parameters; specifically, the to-do item to which the event corresponds. The **CheckBox** control supports a **CheckedChanged** event, which requires a delegate with the following method signature:

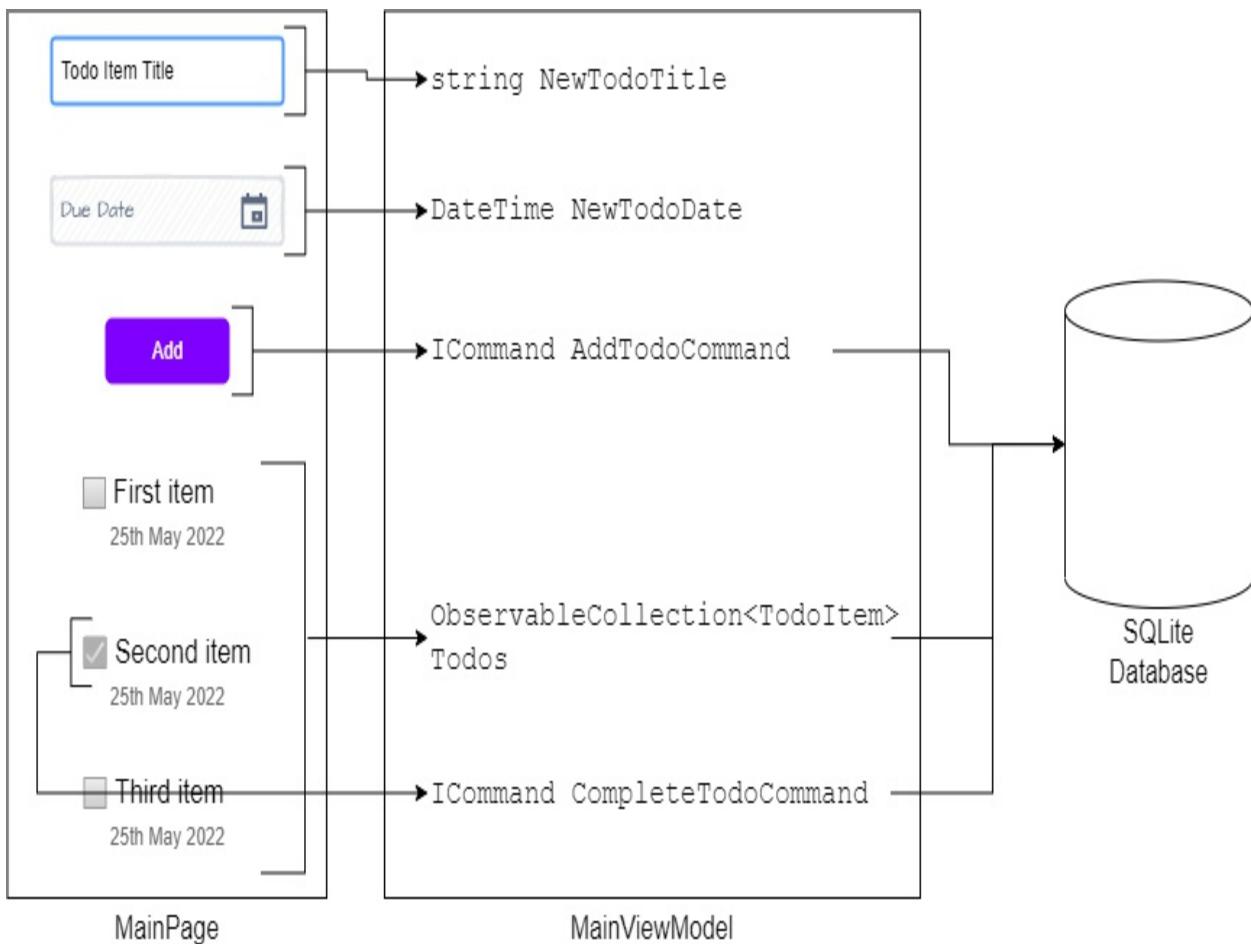
```
void CheckBox_CheckedChanged(Object sender, CheckedChangedEventArgs e)
```

If we try to delegate the `CheckedChanged` event of a `CheckBox` to a method with a different signature (the name of the method is not important), we will get an error. This means that we are limited to receiving two parameters: the sender and the event arguments. The sender in this case is the `CheckBox` itself, which doesn't give us any information about the to-do item, and the event arguments contain a single Boolean value that tells us whether the `CheckBox` is checked or unchecked. The latter is useful, but still doesn't tell us which to-do item is affected.

There are some hacky ways we could work around this problem, but the better approach is to use a *Command instead of an event handler*. `Command` is an implementation of the `ICommand` interface, which defines an `Execute` property - code that will be run when the `ICommand` is invoked. The constructor for `Command` accepts a function that gets assigned to the `Execute` property, which you can declare inline with a lambda expression, or you can pass in a method. A `Command` can accept parameters, which solves our problem of identifying the to-do item, but `CheckBox` doesn't have an `ICommand` property to bind.

Let's start by refactoring MauiTodo. We'll move the code out of the code behind and into a `ViewModel`, which will have a `Command` instead of an event handler. Then we can come back to the problem of a `CheckBox` lacking an `ICommand` property. Figure 9.2 shows the new architecture of MauiTodo, using the MVVM pattern.

Figure 9.2 After we refactor MauiTodo for MVVM, the code behind for MainPage will do nothing except set the binding context to the ViewModel. All of the controls in the UI will be bound to properties in the ViewModel, and the ViewModel will be responsible for communicating with the database. This approach is cleaner and honours the single responsibility principle.



The first step is to create the **ViewModel**. A good convention for naming your ViewModels is to use the name of the View and substitute ViewModel for Page or View in the name. We're creating a ViewModel for our MainPage, so we'll call it MainViewModel.

In your MauiTodo project, create a folder called `viewModels`, and create the class `MainViewModel.cs`.

Our MainViewModel will implement the `INotifyPropertyChanged` interface. `INotifyPropertyChanged` defines an event handler that notifies the UI when a property on the ViewModel has changed. This is necessary when using MVVM; as we are no longer directly manipulating properties on the UI, we need to raise an event to inform the UI that a property has changed, and that the UI should be updated.

In the method that invokes the `PropertyChanged` event, we'll use an attribute

called `CallerMemberName` on the method parameter. This will let us call the method without specifying a property name; instead, when we call it from a property's setter, the name of the property can be automatically inferred. Listing 9.1 shows the boilerplate code for the `MainViewModel.cs` file.

Listing 9.1 The initial code for MainViewModel.cs

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

namespace MauiTodo.ViewModels
{
    public class MainViewModel : INotifyPropertyChanged
    {
        #region INotifyPropertyChanged
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string
        {
            var changed = PropertyChanged;
            if (changed == null)
                return;

            changed?.Invoke(this, new PropertyChangedEventArgs(pr
        }

        public void RaisePropertyChanged(params string[] properti
        {
            foreach (var propertyName in properties)
            {
                PropertyChanged?.Invoke(this, new PropertyChangedE
            }
        }
        #endregion
    }
}
```

Next, let's add the ViewModel's properties and fields. This will be largely the same as what we already have in the `MainPage` code behind, but with a few small differences. The fields will be used to hold private data internal to the ViewModel, but properties must be used for binding. We'll add an `ObservableCollection` of type `TodoItem`, a `string` to hold the title of any new to-do item, a `Datetime` to hold the due date of a new to-do item, and an `ICommand` each to add a new to-do item and mark one as complete.

Listing 9.2 shows this added code, along with the namespaces you need to bring in.

Listing 9.2 The MainViewModel's properties and fields.

```
using MauiToo.Models;
using MauiTodo.Data;
using System.Windows.Input;
using System.Collections.ObjectModel;

...
public ObservableCollection<Todoitem> Todos { get; set; } = new()
public string NewTodoTitle { get; set; }
public DateTime NewTodoDue { get; set; } = DateTime.Now;
public ICommand AddTodoCommand { get; set; }#A
public ICommand CompleteTodoCommand { get; set; }#B
private readonly Database _database;
```

Next up, let's add some methods to the ViewModel. We'll add an `Initialise` method, which will get the list of to-do items from the database and populate the `ObservableCollection`. We'll add a method for adding a new to-do item, which will be largely the same as the method in the code behind, and we'll add a method for marking a to-do item as complete. This method will take a `TodoItem` as a parameter and pass it to the `UpdateTodo` method of the database. The code for these methods is shown in listing 9.3.

Listing 9.3 The methods to add to MainViewModel

```
private async Task Initialise()
{
    var todos = await _database.GetTodos();

    foreach(var todo in todos)
    {
        Todos.Add(todo);
    }
}
```

```

public async Task AddNewTodo()#A
{
    var todo = new Todoitem
    {
        Due = NewTodoDue,
        Title = NewTodoTitle
    };

    var inserted = await _database.AddTodo(todo);

    if (inserted != 0)
    {
        Todos.Add(todo);

        NewTodoTitle = String.Empty;
        NewTodoDue = DateTime.Now;

        RaisePropertyChanged(nameof(NewTodoDue), nameof(NewTodoTi
    }
}

public async Task CompleteTodo(Todoitem todoitem)#C
{
    var completed = await _database.UpdateTodo(todoitem);

    OnPropertyChanged(nameof(Todos));
}

```

Finally, let's add a constructor to `MainViewModel`, which will assign a new instance of the `Database` to the private field. We'll also wire up the two `ICommand` properties by assigning them to new instances of the `Command` type, with their respective methods set as the `Execute` property. For the update method, we can use a `TodoItem` as a type argument. The last step is for the constructor to call the `Initialise` method. As this method is `async`, we'll use the `discard` operator.

`Listing 9.4` shows the constructor for `MainViewModel`.

Listing 9.4 The MainViewModel constructor

```

public MainViewModel()
{

```

```

        _database = new Database();

        AddTodoCommand = new Command(async () => await AddNewTodo());

        CompleteTodoCommand = new Command<TodoItem>(async (item) => a
        _ = Initialise();
    }
}

```

Let's update the code behind now – we're going to remove nearly all the code, as the functionality we're handling here will be moved to the **ViewModel**. **Delete all the properties and methods from MainPage.xaml.cs, leaving only the constructor, and from the constructor, remove all the code other than the InitializeComponent() call from the template.**

Then, in the constructor after the `InitializeComponent()` call, add the following line:

```
BindingContext = new MainViewModel();
```

We learned about the `BindingContext` earlier in Chapter 3, and as `ContentPage` inherits `BindableObject`, it has a `BindingContext`. Here we are setting it to be a new instance of `MainViewModel`. This means that anywhere we declare a binding to a source property, it will be a property on this object.

The last step in our MVVM refactor is to update the UI to use the new binding context. We'll add a property binding for the `Entry` and `DatePicker` controls and replace the event handler on the `Button` with a `Command`. Finally, we'll set a binding for the `ItemsSource` on the `CollectionView`, seeing as we are no longer assigning that in the code behind.

Update your `MainPage.xaml` file to match listing 9.5.

Listing 9.5 MainPage.xaml with MVVM bindings

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MauiTodo.MainPage"
    BackgroundColor="{DynamicResource SecondaryColor}">
```

```

<Grid RowDefinitions="1*, 1*, 1*, 1*, 8*"
      MaximumWidthRequest="400"
      Padding="20">

    <Label Grid.Row="0"
          Text="Maui Todo"
          HorizontalTextAlignment="Center"
          FontSize="Title"/>

    <Entry Grid.Row="1"
           HorizontalOptions="Center"
           Placeholder="Enter a title"
           WidthRequest="300"
           Text="{Binding NewTodoTitle}"/> #A

    <DatePicker Grid.Row="2"
                WidthRequest="300"
                HorizontalOptions="Center"
                Date="{Binding NewTodoDue}"/> #B

    <Button Grid.Row="3"
            Text="Add"
            WidthRequest="100"
            HeightRequest="50"
            HorizontalOptions="Center"
            Command="{Binding AddTodoCommand}"/> #C

    <CollectionView Grid.Row="4"
                    ItemsSource="{Binding Todos}"> #D
      <CollectionView.ItemTemplate>
        <DataTemplate>
          <SwipeView>
            <SwipeView.LeftItems>
              <SwipeItems Mode="Reveal">
                <SwipeItem Text="Delete"
                           IconImageSource="delete"
                           BackgroundColor="Tomato"
                           TextColor="White" />
              </SwipeItems> #E
            </SwipeView.LeftItems>

            <SwipeView.RightItems>
              <SwipeItems Mode="Reveal">
                <SwipeItem Text="Done"
                           IconImageSource="check"
                           BackgroundColor="LimeGreen" />
              </SwipeItems>
            </SwipeView.RightItems>
          </SwipeView>
        </DataTemplate>
      </CollectionView.ItemTemplate>
    </CollectionView>

```

```

<Border Stroke="{StaticResource PrimaryColor}"
        StrokeThickness="3"
        StrokeShape="RoundRectangle 10"
        Padding="5"
        Margin="0,10">
    <Border.Shadow>
        <Shadow Brush="Black"
                Offset="20, 20"
                Radius="40"
                Opacity="0.8"/>
    </Border.Shadow>
    <Grid WidthRequest="325"
          ColumnDefinitions="1*, 5*"
          RowDefinitions="Auto, 25"
          x:Name="TodoItem">

        <CheckBox VerticalOptions="Center"
                  HorizontalOptions="Center"
                  Grid.Column="0"
                  Grid.Row="0"
                  IsChecked="{Binding Done}" />

        <Label Text="{Binding Title}" #FontFamily="Futura"
               FontAttributes="Bold"
               LineBreakMode="WordWrap"
               HorizontalOptions="StartAndEnd"
               FontSize="Medium"
               Grid.Row="0"
               Grid.Column="1"/>

        <Label Text="{Binding Due, StringFormat='MM/dd/yyyy'}"
               VerticalOptions="End"
               Grid.Column="1"
               Grid.Row="1"/>
    </Grid>
</Border>
</SwipeView>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
</Grid>
</ContentPage>

```

At this point you should be able to run the MauiTodo app and see it working the same way as before. While the functionality is the same, we've now got a much cleaner app; our business logic and UI are cleanly separated, and this

enables us to implement features that would be difficult or messy to do before, like setting the `Done` state of a to-do item from the UI, which we will do in the next section, and making the app easier to maintain overall.

9.2 Use behaviors to augment your controls

We've still got one last problem to solve with our to-do app, and that is finding a way to mark to-do items as complete. The model has a `Done` property, and the UI has a `Checkbox`, but we still need to figure out a way to connect them.

The problem is that the `Checkbox` control has a `CheckedChanged` event which can delegate to an event handler, but it doesn't let us send any custom parameters. We've looked at `commands` which we can use in MVVM – these do allow parameters, but the `checkbox` control doesn't have a `Command` property that can be bound.

We can solve this problem by using a **behavior**. Behaviors let us extend the functionality of UI controls, and we do this by attaching a behavior to a control, rather than subclassing the control to our own type. Subclassing controls is useful in some cases, but unnecessary here.

A behavior can be attached to any property or method of a control, so we'll attach a behavior to the `CheckedChanged` event of the `Checkbox` and fire a `Command` in response. In the listing in the last section, we recapped how in a `CollectionView`, the item itself is the binding context, which means we have access to the to-do item the `checkbox` corresponds to. Depending on our use case, we could send the `Id` of the to-do item to a method that would mark it as done. In our case, we're going to send the whole to-do item and pass it to the `Update` method in the database. This approach fits our requirements and can also be re-used if we decide to add edit functionality in the future, rather than writing a bespoke method just for one use case.

Extending a control in this way – calling a `Command` in response to an event being triggered – is a common requirement in MVVM. So common, in fact, that there's a ready built Event to Command Behavior that we can use in the **.NET MAUI Community Toolkit**. The Community Toolkit is a set of open-

source add-ons for .NET MAUI that are, as the name suggests, contributed by the community. These are things that meet requirements that are so ubiquitous that it makes sense to gather them all in one place for people to use in their .NET MAUI apps, even though they are not part of the core .NET MAUI product.

The Community Toolkit is an invaluable collection of features that you will likely come to depend on in your .NET MAUI apps, and I encourage you to explore it here: <https://learn.microsoft.com/dotnet/communitytoolkit/maui>. But for now, let's focus on the Event to Command Behavior.

The first thing you need to do is install the `CommunityToolkit.Maui` NuGet package into the `MauiTodo` app. Many of the features in the toolkit require an initialization line to be added to the host builder in `MauiProgram.cs`. This is covered in the “Get Started” section of the documentation, and as you will likely use this toolkit in your apps, you should get to grips with this. But, for simply importing XAML namespaces, this is not required. We want to use the `EventToCommandBehavior`, which is in the `CommunityToolkit.Maui.Behaviors` namespace. Add this namespace to your `MainPage.xaml` file by adding the following line to the `ContentPage` tag:

```
xmlns:behaviors="clr-namespace:CommunityToolkit.Maui.Behaviors;as
```

This brings in the namespace and gives it an XML namespace reference of behaviors. With this in place, we can reference this in our XAML code. Update the `CheckBox` tag to match Listing 9.6.

Listing 9.6 CheckBox in MainPage.xaml

```
<CheckBox IsChecked="{Binding Done, Mode=TwoWay}"  
         VerticalOptions="Center"  
         HorizontalOptions="Center"  
         Grid.Column="0"  
         Grid.Row="0">#A  
    <CheckBox.Behaviors>  
        <behaviors:EventToCommandBehavior #B  
          EventName="CheckedChanged" #C  
          Command="{Binding Source={x:Reference TodoPage}, #D  
                            Path=BindingContext.CompleteTodoCommand}" #E  
          CommandParameter="{Binding .}"/#F
```

```
</CheckBox.Behaviors>  
</CheckBox>
```

Run the MauiTodo app now. You can add some to-do items, and you can check the CheckBox for each to-do item. You could of course do this before, but now, if you add a breakpoint in the Update method of the database, you'll see it gets hit when you check or uncheck a to-do item. Better yet, if you quit the MauiTodo app, and re-open it, you'll notice that any items that you checked to mark as complete, will still be checked now. This is because when we check the box and trigger the CheckedChanged event, the EventToCommand behavior is responding and calling the CompleteTodoCommand in the ViewModel. This in turn calls an Update method on the database which persists our change to disk.

By using a Behavior, we've added functionality to a Control, without having to subclass the Control or build our own, that enables us to abstract functionality out of our View and into our ViewModel, and solved a critical outstanding problem with the MauiTodo app.

Exercise

Back in Chapter 4 we introduced SwipeView to the MauiTodo app, but it only showed a message confirming our intention (to delete or complete a to-do item) rather than doing the action.

In section 9.1 we deleted the SwipeItem's invoked event handler from the MainPage code behind, so if you run the app now, you'll get an error as we are still referring to this event handler in the XAML.

Let's complete the swipe feature in MauiTodo.

1. Use the SwipeView's Command property to bind to the CompleteTodoCommand in the ViewModel
2. Add a DeleteTodoCommand to the ViewModel, and use the SwipeView's Command property to bind to it

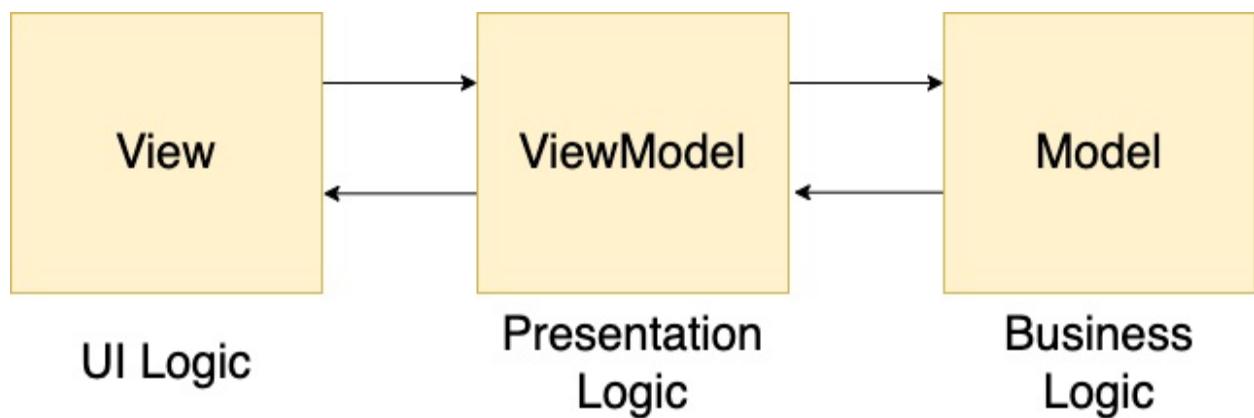
Once you complete this exercise, you'll be able to add to-do items, mark them as complete (either by checking the box or using the SwipeView) or

delete them (using the `SwipeView`).

9.3 What is MVVM?

MVVM is a software design pattern that promotes organization of code by keeping business logic, presentation logic and UI separate, as summarised in figure 9.3.

Figure 9.3 With the MVVM pattern, UI logic, which includes layouts and UI behavior like animation and text formatting, belongs in the view. Presentation Logic, which includes UI state such as the values of properties in the UI and logic for responding to user actions, belongs in the ViewModel. Business logic, such as rules for processing user input, or communicating with an API, belongs in the Model.



Using the MVVM pattern, we put the single-responsibility principle into practice by keeping layout and UI behavior in Views, UI state and presentational logic in ViewModels, and business logic in the Model. UI behavior might include things like changing the color of a button or label, or animating a UI element. Presentational logic is essentially UI state, for example the value entered into an `Entry`. Business logic is how the real-world scenario is modelled in code. This would include models representing real-world objects the app deals with, e.g., a model representing a car might have properties for make, model and year, and methods for altering the object's state. Business logic also includes what we do with our models. Examples might include methods for querying a database or API for a list of cars based on certain criteria. In essence, the Model is the problem domain represented in code.

9.3.1 The Model

You will no doubt have come across the term ‘model’ before, but in MVVM it has a specific meaning which may (or may not) differ from what you are used to. The word model is sometimes used to refer to a single class, for example a car class might contain properties that represent real properties of a car (like make, model and year), and as such may be considered to model a car.

This is a narrower perspective than in MVVM, where the *Model* refers to the model of your entire problem domain, rather than a specific class. This would include classes that model real-world objects, like the car example, as well as other classes containing domain logic. These might include business rules, or logic for communicating with a database or REST API. These kinds of classes are sometimes called the service layer, or any variation of application layer or infrastructure layer. In MVVM, individual models (like the car) and other business logic (like the services) together make up the Model.

Model vs model: Disambiguation

The word model is often used in software development and computer science (and many other fields). In this book, if you see the word capitalized (Model), then I am specifically referring to the Model component of the MVVM pattern. If you see model with a lowercase first letter, then I am using the term with a different meaning which you can infer from the context. The Car class mentioned above is a model, but not a Model.

Classes like Car are often referred to as entities, but to add to the confusion, a common convention in MVVM projects is to put entities in a folder called Models, and classes with business logic methods in a folder called Services.

I follow that convention in this book, as I find it easy enough to distinguish between MVVM as an architectural pattern and the organization of files and folders in my solution. You don’t have to follow that convention in your own projects, but you’ll have to get to grips with that to follow the examples in the book. It will be useful for you to get used to this approach, as you’ll likely encounter it in projects you work on or view online.

The Model is comprised of these classes that define objects and functionality that represent the problem domain. In this sense, you can think of the Model as being the same as a domain model in any other .NET application.

9.3.2 The View

We've covered Views in detail in earlier chapters, and Views, as we described them there, fit the same definition we use in the MVVM pattern. To recap, Views come in three flavours: Pages, Layouts and Controls. Together these provide the definition of *how* your app's information is displayed to a user, which we refer to as UI logic. UI logic is essentially comprised of layout and behavior. Layout is the definition of what elements go where on screen, and behavior is what those elements do.

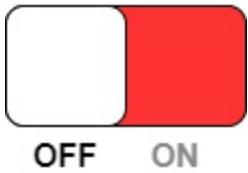
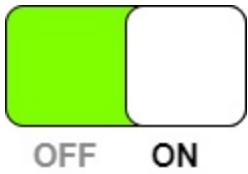
For example, a Page that contains a Grid, inside which are some Labels, an Entry and a Button are layout. An animation that grows and shrinks different Labels based on certain conditions is behavior.

9.3.3 The ViewModel

The *ViewModel* is the heart of the MVVM pattern, as it contains the core logic that controls what the user will see and interact with. ViewModels contain presentational logic, which differs from UI logic in some important ways. Where the View defines *how* things are displayed, the ViewModel defines *what* is displayed.

For example, a Switch in a view can have a state of on or off (see figure 9.4). This state is presentational logic and belongs in the ViewModel. The color of the Switch is UI logic and should be defined in the view.

Figure 9.4 A Switch control. In the top example, the Switch is in the ON position (its state), and its background is green (its behavior). In the bottom example, the Switch is in the OFF position and its background is red. The Switch's state should be managed by the ViewModel, and its behavior should be managed by the View.



If you want your switch to appear red when in the off state, and green when in the on state, this is UI logic (technically UI behavior), and while the *state* of the Switch (on/off) should be maintained in the ViewModel, the behavior of that Switch, which changes in response to its state, belongs in the View.

ViewModel vs view model: disambiguation

You may be familiar with the term ‘view model’ (and you’ll see me use it in this chapter). It’s common to refer to a view model outside of MVVM, and it’s important to know that in MVVM, the term ViewModel has a different, and very specific, meaning.

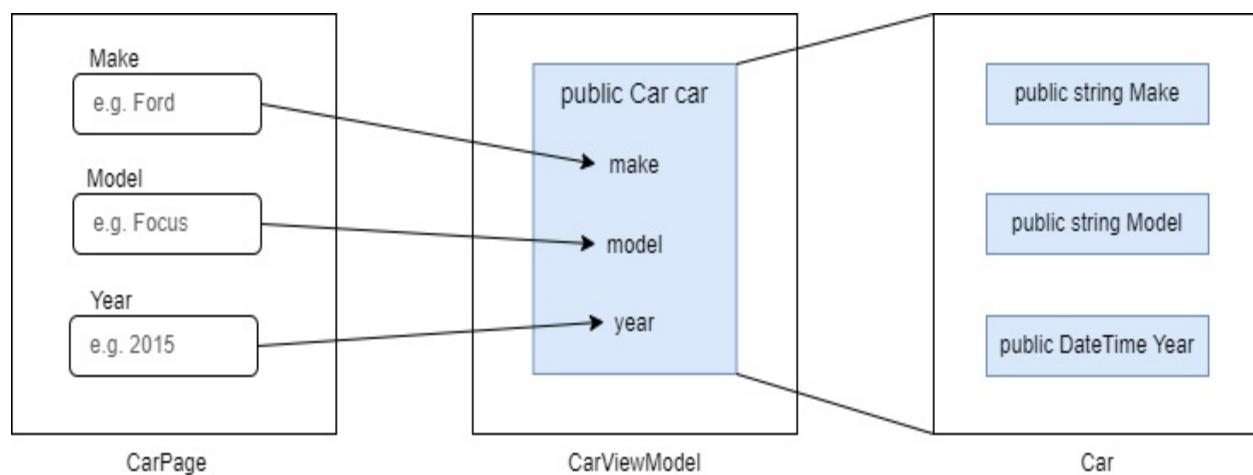
Outside of MVVM, the term view model can be fluid, but it usually refers to a data transfer object (DTO) that encapsulates all data required for a view, rather than for a specific model or entity. For example, in a web app that shows a sales leaderboard, you might have a DTO that represents an individual salesperson, and another that represents a monthly total, and another for a monthly average, etc. Instead of having your web view call each of these individually, the API can return a view model that contains all the data the view needs to render the leaderboard so that a single API call is all that’s required. (Incidentally, GraphQL was designed to solve this problem; this is way outside the scope of this book but check out *GraphQL in Action* by Samer Buna, Manning 2021, to find out more about this exciting topic).

In MVVM the term ViewModel, which I will always write as a single word in PascalCase, specifically means a class that represents the state of, and provides functionality for, a View. The key difference between an MVVM

ViewModel and a view model as a DTO is that a ViewModel contains functionality. A DTO, by definition, cannot.

Figure 9.5 shows a more complete example of the relationship between a View, ViewModel and Model.

Figure 9.5 A View called CarPage has controls that let the user enter the make, model and year of a car. The values for these controls are not stored in properties on the View. Instead, Databinding is used to bind the values of those controls to properties in a ViewModel. Specifically, the ViewModel has a property called car of type Car, and the Car type has properties for make, model and year, and the UI controls are bound to these properties of the Car instance in the ViewModel.



9.3.4 Binding from Views to ViewModels

The View uses Databinding to connect to properties in the ViewModel. This means that the ViewModel holds the values of controls in the View, and the View is only responsible for rendering, rather than storing or processing, information.

ViewModels also contain all the code for interacting with the Model, which keeps this logic out of the View. Views can of course do a lot more than just display data or allow data to be edited. Many of the controls we use in .NET MAUI have events which we can write event handlers for; the simplest example being a Button. We've seen before that you can create an event handler to respond to a click, but this method is delegated rather than data bound. This means that it must exist in the code behind file.

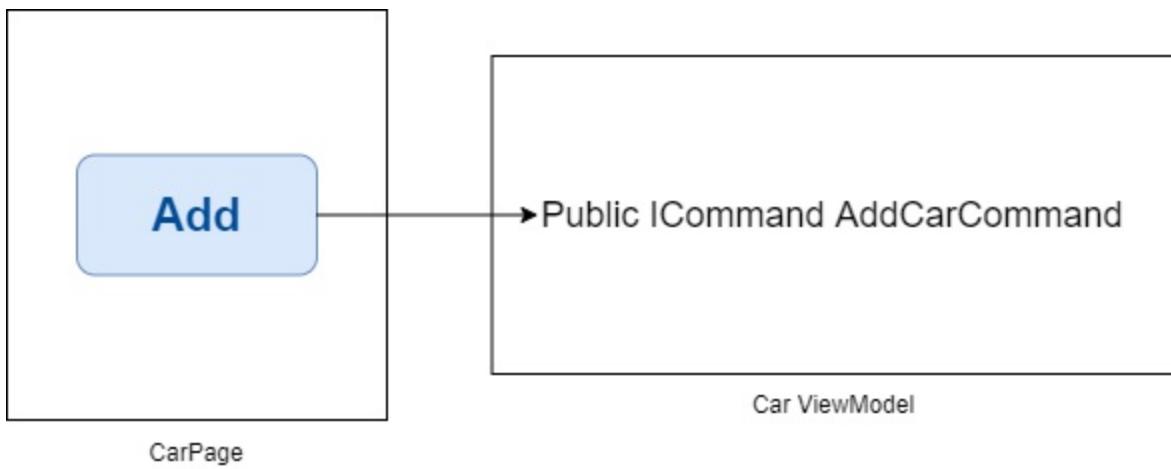
There is a way of working around this, which is to delegate the event handler in code rather than in XAML. For example, you could create a method with the right signature in your ViewModel, and then in your View constructor, you can use the following code:

```
MyButton.Clicked += MyViewModel.MyEventHandler;
```

But this introduces several problems. First, when we write an event handler for a click event in a code-behind, that code also has access to other UI properties. As an example, in the FindMe app from chapter 3, when the user clicks a Button, we retrieve their name from an Entry in the UI and then use it in the message we share along with location. If this method were in a ViewModel, it would have no access to the Entry to retrieve that name. Another problem, that we solved with MVVM for MauiTodo, is that we often can't pass meaningful parameters with an event handler.

Instead of delegating events that are raised in the View to handlers in ViewModels, in .NET MAUI we use the principle of *commanding*, which solves these problems and gives us more flexibility. Several built-in controls in .NET MAUI have a **Command** property, which is bindable to a property on a ViewModel of type **ICommand** (see figure 9.6). .NET MAUI also has built in implementations of the **ICommand** interface, and by including these in our ViewModels and binding to them in our Views, we can take all the presentational logic out of our Views, leaving just the necessary UI logic.

Figure 9.6 The **Command** property of the **Button** control in the View is bound to the **AddCarCommand** property of the **ViewModel**. When the **Button** is tapped, the **ICommand** in the **ViewModel** is called, and any logic defining if and how it executes code is all handled by the **ViewModel**.



After moving logic from our View to our ViewModel, we also need a way for ViewModels to update controls in the View when the values of bindable properties change. Without MVVM, we could set the values of properties directly on UI controls; now that we are using MVVM we can't do this.

We've seen before how we can use data binding to bind properties of UI control to properties of ViewModels. To make this work both ways (i.e., to update the UI when the value of a bound property changes), we need to implement the **IPropertyChanged** interface from the System.ComponentModel namespace in our ViewModel.

IPropertyChanged defines a **PropertyChangedEventHandler** event called **PropertyChanged**.

Note

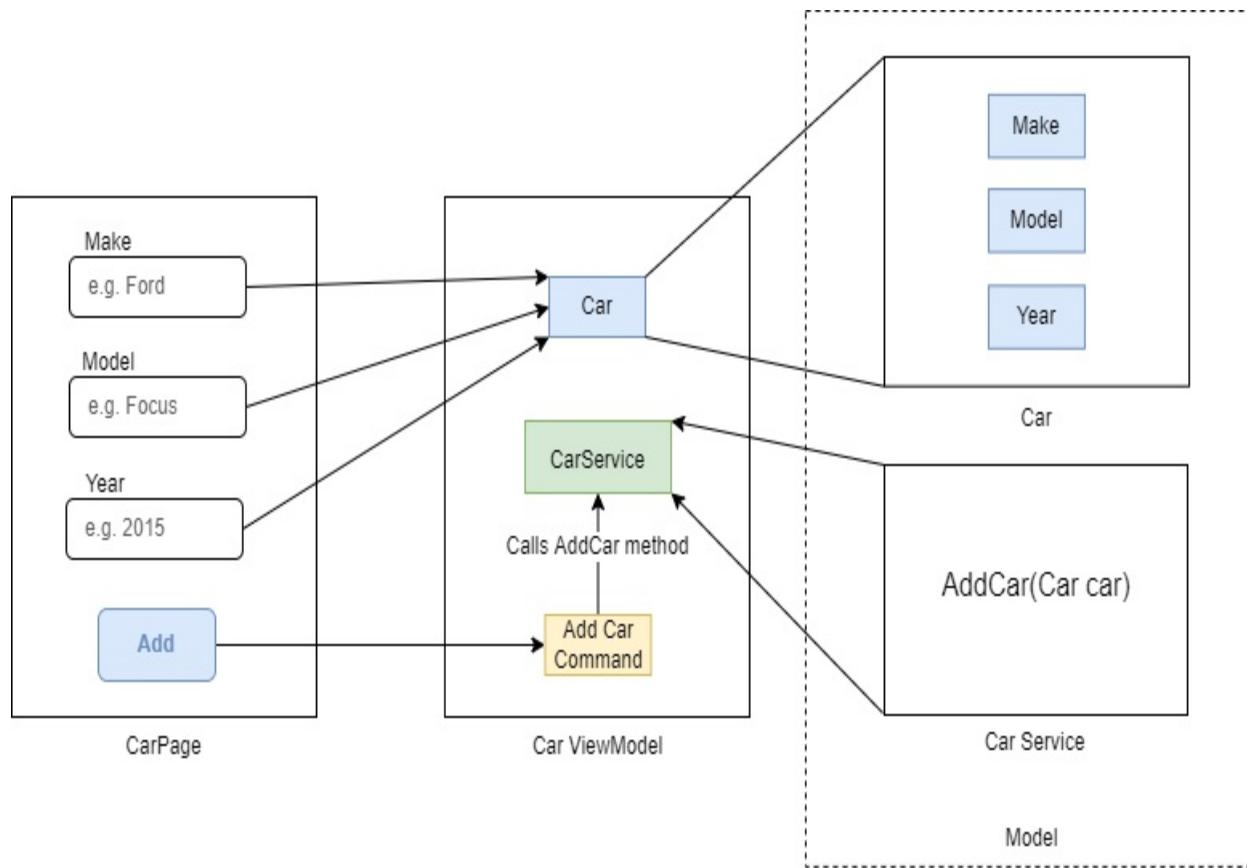
All Pages in .NET MAUI implement **IPropertyChanged**, which is how we were able to use it in MauiMovies.

When a View binds to a ViewModel, it subscribes to **PropertyChanged** events. When an event is raised for a property that the View is bound to, it will update the corresponding value in the View.

9.3.5 Putting it all together

In figure 9.7 we can see an example architecture of an app that interacts with a database of cars, that is organized using the MVVM pattern.

Figure 9.7 In the MVVM pattern, the View only contains UI logic. Controls are bound to properties on a ViewModel, and rather than event handlers, commands are used to trigger actions. The ViewModel maintains the state of the View, by exposing properties for the UI to bind to and commands for the UI to call. The Model contains all the business logic, and the ViewModel interacts with the Model. In this example, a CarService from the Model is injected into the ViewModel. A Button in the View is used to trigger a command in the ViewModel. The ViewModel contains a Car object, which has properties bound in the UI. The Add Car Command calls the AddCar method in the injected service and passes in the ViewModel's Car property as a parameter.



We can see that by following the MVVM pattern, we gain the following advantages:

- **Honours the single-responsibility principle.** The View is only responsible for UI. The ViewModel is only responsible for View state. The classes that make up the Model each serve their own specific function.
- **Easier to maintain.** Because each of these components are loosely coupled, we can change any one of them with little or no impact on any other. Designers or UI developers can work independently of developers.

working on the presentation logic or business logic. Problems are more easily isolated and new features are easier to implement.

- **Easier to test.** Because the ViewModel is code only, we can write unit tests that can validate nearly every function of the UI. In fact, we can test any logic we like that exists outside of the View, and in MVVM, that's everything except layout and UI behavior.

Do I always have to use the MVVM pattern?

The MVVM pattern is a tool that you will learn to use when adds value to your work. In some cases, particularly with trivial, one-screen apps, the MVVM pattern is not necessary. Even for complex apps, there are alternatives to MVVM, in particular the Model-View-Update (MVU) pattern which is quite popular. MVU is a state-based pattern with an immutable UI and is a good fit if you prefer to define your UI in C# code rather than markup.

You can choose to use MVVM, MVU or neither in your .NET MAUI apps. Which one you use will be up to you and your team, although while MVU is considered a ‘first class citizen’ in .NET MAUI, it is not supported out of the box with first party tooling. That is why MVVM is still considered by many to be the standard pattern for use in .NET MAUI, and why it is the pattern used in this book.

If you are familiar with the MVU pattern and are interested in using it in .NET MAUI, check out Gerald Versluis’ getting started video:

<https://youtu.be/52RmT2MIFzg>.

We’ll complete MauiStockTake using the MVVM pattern, which will give you a better appreciation of its use. However, the MVVM pattern is an important topic, and one you should endeavour to learn in more depth as you progress through your .NET MAUI journey. As you do, you’ll start to gain an intuitive understanding of what belongs where in the MVVM pattern. In the meantime, the following table provides some examples to help you understand where to place the different components of your app.

Table 9.1 In the MVVM pattern, UI definitions and behavior belong in the View, presentational logic belongs in the ViewModel, and business logic belongs in the Model

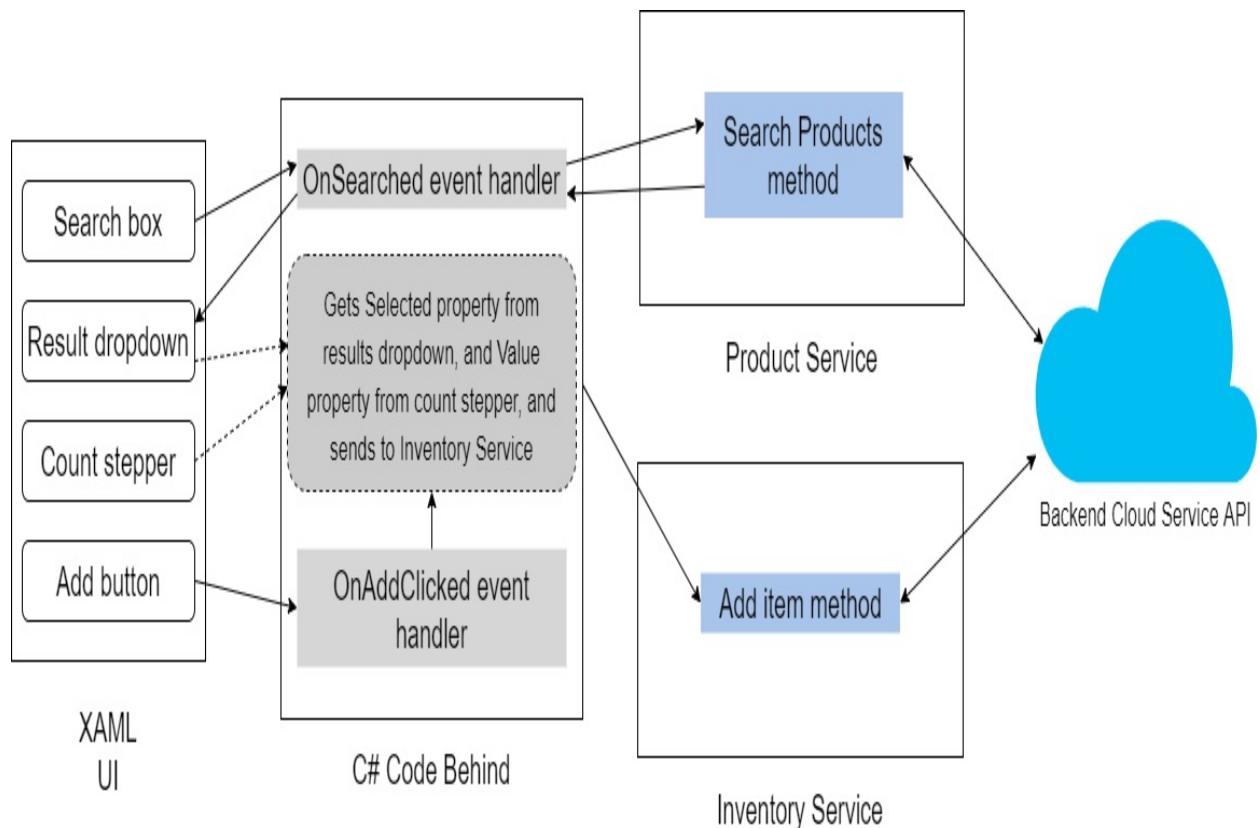
Logical Function	Where it belongs	Example
UI definition	In the View	XAML markup defining a Button or Label
UI behavior	In the view	Code for animating a button in response to a tap
Presentational logic (UI state)	In the ViewModel	Properties representing controls displayed in the UI, e.g., a Name string that a Label in the UI is bound to, that gets data from a Name property on a User object in the Model
Business logic	In the Model	Properties of a car in a car app (e.g., make, model, year, etc.) Or Code that calls a search method in a repository or API, passing in user input as a search parameter.

9.4 The MauiStockTake app without MVVM

It's entirely possible to build the MAUI stock take app without MVVM. In fact, we've built several apps so far throughout this book without using the MVVM pattern (although we've started to use some of the features that enable it, such as data binding and service abstraction). These apps have been

trivial compared to MauiStockTake, so building them without MVVM hasn't presented us with any problems (apart from the CheckBox in MauiTodo). Without MVVM, the architecture of the MauiStockTake looks something like figure 9.8.

Figure 9.8 Without MVVM, services are injected into the UI. Business logic in the UI gathers data from UI controls and constructs requests for services.



This would work and would give Mildred the functionality she needs. But there are some problems with this approach, which you may have spotted:

- **Violates the single-responsibility principle.** The UI should be responsible for UI only. But in this approach, the UI is managing application state and executing business logic.
- **Difficult to maintain.** This architecture makes it difficult to identify and fix errors, and makes it difficult to modify the functionality or add new features. There is tight coupling because business logic is included directly in the UI. Extending logical functionality or changing the UI directly impact each other.

- **Difficult to test.** Business logic in the UI is almost impossible to test in any way other than manually.

You may have spotted other problems with this approach too. We can solve these problems by adopting the MVVM pattern, which will make it easier for us to write longer lasting apps with maintainable and adaptive code.

MVVM for SOLID apps

SOLID is a set of object-oriented programming (OOP) principles that help to build clean and maintainable code. SOLID is an acronym that summarises the principles set out below.

Single responsibility principle: The single responsibility principle (SRP) states that any class or method should be responsible for one thing and one thing only. All parts of that class or method should be aligned with that purpose.

Open/closed principle: The open/closed principle states that classes and methods should be open to extension but closed to modification. This means that you extend existing functionality in response to new requirements, rather than changing it. In practical terms, this is achieved using abstract classes and interfaces. Interfaces and base classes should not be modified (closed), but new implementations or derived types can be added freely (open).

Liskov substitution principle: The Liskov substitution principle states that any derived type can stand in for its parent type. Essentially this means that if you inherit a base class, your class must not violate the contract provided by the base class. A derived class must be able to be used anywhere a base class could be.

Interface segregation principle: The interface segregation principle states that an interface (a definition of a dependency) must be tightly focused on only the required functionality for that dependency. For example, if you have a class that needs a `Sum()` method, the dependency for that class should declare a `Sum()` method only, and no other methods like `Subtract()` or `Multiply()`. In this simple example, you may in fact also require those other methods in the consuming class, in which case it would make sense to define

an `ICalculator` interface that declares all of the basic arithmetic methods. However, remember that C# allows multiple interface inheritance, so you can still write a `calculator` class that implements all your arithmetic operations defined on different interfaces. This may not be necessary for a calculator, but in more complex scenarios, it's best to keep interfaces focused on the specific dependency that they define.

Dependency Inversion principle: The dependency inversion principle is fundamentally what enables us to write loosely coupled code. It states that we invert dependencies by defining them where we need them, rather than where they are provided. The `Sum()` example above illustrates this; a consuming class knows that it needs a `Sum()` method, but doesn't necessarily know (or care) about the `calculator` class. This means that the implementation of the `Sum()` method can be changed without any impact on the class that consumes it.

These are somewhat simplistic and naïve explanations, but hopefully you are familiar with these concepts already. If not, it's well worth investing some time in gaining an understanding and appreciation for them.

As we progress through this chapter, you will see how using the MVVM pattern helps us to write SOLID code.

Later in this chapter, we'll finish the inventory feature in MauiStockTake using the MVVM pattern, and see how it addresses these problems.

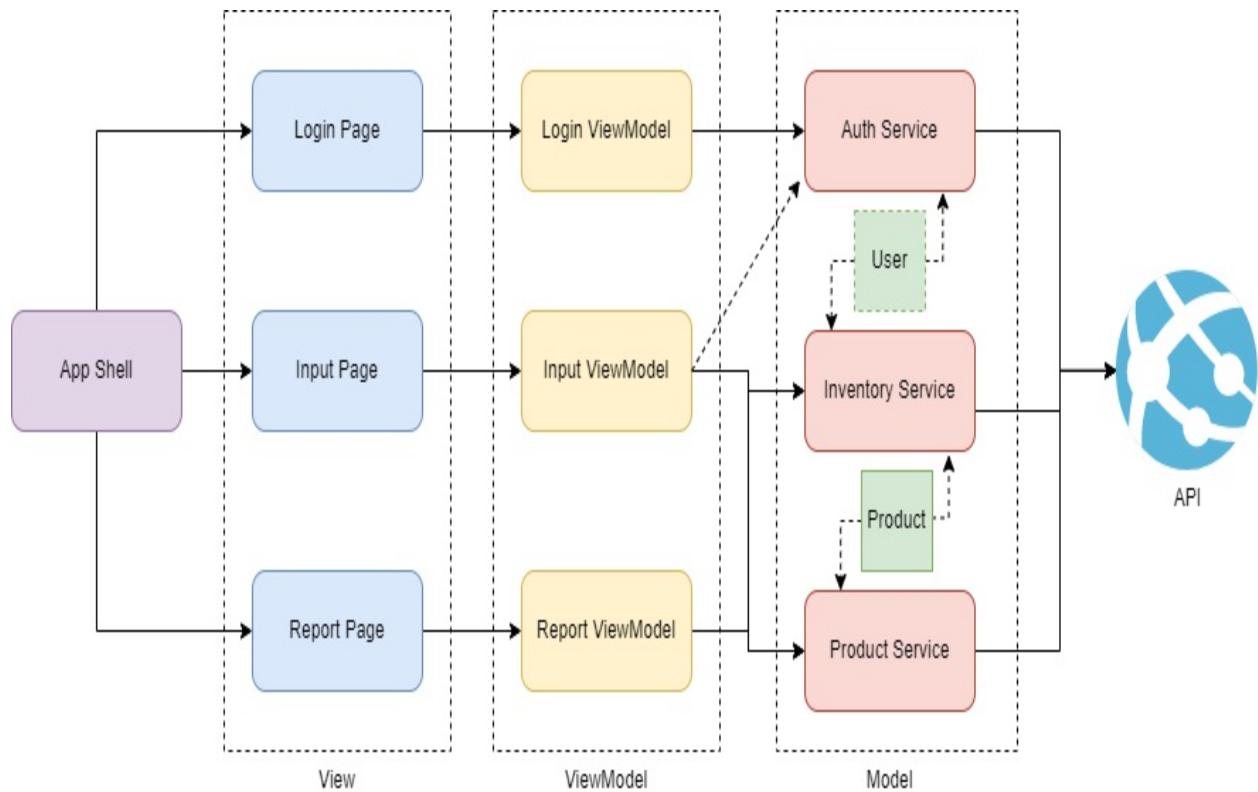
9.5 The MauiStockTake app in MVVM

Earlier in this chapter we looked at what the MauiStockTake app looks like without MVVM. Figure 9.x8 shows how the InputPage would work without MVVM, and we saw some of the problems this approach would cause.

Now that we know about the MVVM pattern, and how it can help us avoid some of the pitfalls we encounter without it, let's look at what the MauiStockTake app looks like with MVVM. In figure 9.9 the app has been designed using the MVVM pattern, and we can see that UI logic (View), presentation logic (ViewModel) and business logic (Model) are all cleanly

separated

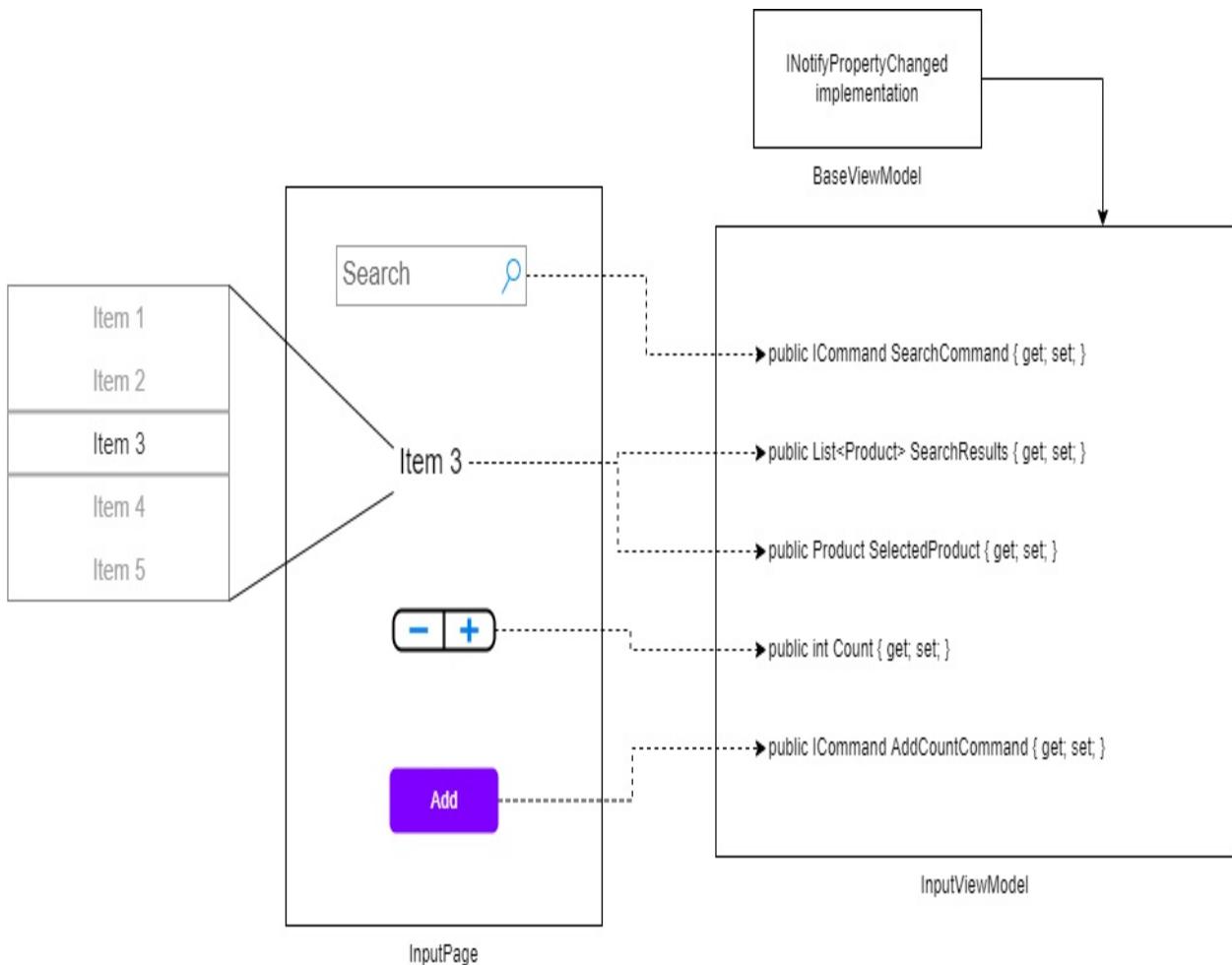
Figure 9.9 The high-level architecture of the MauiStockTake app using the MVVM design pattern. The App Shell defines the overall structure of the app. The LoginPage, InputPage and ReportPage each have a corresponding ViewModel. The Model consists of services containing business logic and individual models like Product and User (there is one missing for stock count, which we will add later). These services interact with the web API where the data will be aggregated.



This high-level app architecture is a good starting point, and as the app grows and changes, we can ensure that anything we add or change is aligned with these design principles.

At a more granular level, figure 9.10 shows the View and ViewModel design for the input workflow.

Figure 9.10 The InputPage with its associated ViewModel in the MVVM pattern. The InputViewModel inherits a common BaseViewModel which contains the INotifyPropertyChanged implementation, and exposes public properties that controls in the corresponding View are bound to.



When we looked at the MauiStockTake architecture without MVVM at the start of this section, we saw that the UI contained nearly all the logic for the app. There was some functionality that was in services, but the UI was doing the heavy lifting.

In figure 9.9 we've changed this around. The code behind for the XAML files will do nothing other than set the binding context, and the UI controls will simply bind to a ViewModel.

The `InputPage` has a `SearchBar` where the user can look for the product they want to inventory. The `SearchCommand` property of the `SearchBar` is bound to an `ICommand` property of the `ViewModel` which will execute the search using a method in the `ProductService`.

The results of that search are put in an `ObservableCollection` which a `Picker` in the View is bound to. The user can select an item from the `CollectionView`, and the `SelectedItem` property of the `CollectionView` is bound to a `SelectedProduct` property on the `ViewModel`. A `Stepper` on the View can be used to indicate how many of the item the user has counted, and this is bound to a `Count` property on the `ViewModel`.

When the user is ready to submit their count, they can click a `Button` that has its `Command` property bound to an `ICommand` on the `ViewModel`. The `ICommand` will call a method in the `IInventoryService` which will send the result back to the API.

This meets the core requirements of the app; we also need reporting, but we'll come back to this in the next chapter. Let's get started building the first prototype of the `MauiStockTake` app.

9.5.1 The Model

We started to describe the Model of the `MauiStockTake` app in chapter 7, in the table where we looked at the problem areas. We saw this extrapolated in figure 9.9, where we can see that the Model for the app consists of the Authentication Service, the Inventory Service, the Product Service, and their associated types.

In `MauiTodo`, the `TodoItem` and the Database comprise the Model. In `MauiStockTake`, the Model is made up of the services in the `MauiStockTake.Client` project and their associated DTOs in the `Shared` project, as well as the `AuthService` we created in the `MauiStockTake.UI` project.

The following table shows how the Model represents the problem areas that the `MauiStockTake` app solves.

Table 9.2 In MauiStockTake, the Model is comprised of services and DTOs that model the problem areas the app is designed to address. For products, the ProductDto is the type that represents actual products, and the ProductService has business logic for working with the ProductDto. For inventory, the InventoryService has business logic for working with the StockCountDto and InventoryItemDto. The AuthService has business logic for dealing with users.

Problem Area	Type(s)	Service
Products	ProductDto	ProductService
Inventory	StockCountDto, InventoryItemDto	InventoryService
Staff	User	AuthService

When you're writing your own apps, you will create models and services that represent the real-world problem your app addresses. But in MauiStockTake, the Model has already been created.

9.5.2 The ViewModel

In this section we're going to create the ViewModel for the `InputPage`. Just as we did in `MauiTodo`, the `InputViewModel` will handle all the state and logic for the `InputPage`, which will have the `InputViewModel` set as its binding context. The `InputPage` will only be responsible for displaying the UI.

The `InputViewModel` will have some properties and methods that are specific to the `InputPage`, but it also has some requirements which will be common to any page in our app. For example, it will need an implementation of `INotifyPropertyChanged` (as we had in the `MauiTodo MainViewModel`), as well as a title, a loading indicator, and an instance of `INavigation`.

NOTE

For Shell, you don't need a reference to the app's navigation stack, as you can call `Shell.Current.GotoAsync()` from anywhere in your app. But you need it for any other navigation paradigm (even in a Shell app, when pushing modal pages for example).

Rather than replicate these methods and properties, we'll implement them in a base ViewModel that other ViewModels can inherit.

Creating the BaseViewModel

We need to implement the `IPropertyChanged` interface in our ViewModels so that we can update the UI in response to changes in the state of the ViewModel. In figure 9.10, we saw that in the MauiStockTake app, we'll implement a base class so that this implementation can be reused in multiple ViewModels.

Create a folder in the `MauiStockTake.UI` project called `viewModels`, and add a new file called `BaseViewModel.cs`. We'll make the `BaseViewModel` class implement the `IPropertyChanged` interface, so that all other ViewModels can derive from the `BaseViewModel`, and we only have to write the implementation once.

Update the `BaseViewModel.cs` file to match the code in listing 9.7.

Listing 9.7 BaseViewModel.cs

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

namespace MauiStockTake.UI.ViewModels;

public class BaseViewModel : IPropertyChanged #A
{
    public event PropertyChangedEventHandler PropertyChanged; #B

    public void OnPropertyChanged([CallerMemberName] string prop
    {
        var changed = PropertyChanged; #D
        if (changed == null)
            return;

        changed.Invoke(this, new PropertyChangedEventArgs(prop));
    }
}
```

The `BaseViewModel` give us an implementation of `IPropertyChanged`,

so we can bind a View to any ViewModel that inherits this base class and invoke the handler to notify the UI of changes. We can take our `BaseViewModel` beyond a simple `INotifyPropertyChanged` implementation though. There is some other common functionality we expect to need in all our pages. In `MauiStockTake`, every page will have a title, so we can add a `Title` property to the base ViewModel. We will also need a loading indicator, so we can add a property for this too. We'll need to add backing fields for these and call `OnPropertyChanged` in the setters for the properties.

We will also want to add an `INavigation` property to our `BaseViewModel`. Navigation and control flow is presentation logic, so we want this to be managed by the ViewModel rather than the View.

You will see these properties (`Title`, `Navigation` and `IsLoading`) in base ViewModels in a lot of projects, and in your own apps you may find there are other common properties you want to implement too.

Listing 9.8 shows the properties to add to the `BaseViewModel` class.

Listing 9.8 The BaseViewModel common properties

```
public class BaseViewModel : INotifyPropertyChanged
{
    private string _title;
    public string Title { get => _title; set { _title = value; 0 }

    private bool _isLoading;
    public bool IsLoading { get => _isLoading; set { _isLoading

    public INavigation Navigation { get; set; }

    ...
}
```

Tip

Keep this `BaseViewModel` handy as you can refer to it for any of your apps. You can also use the MVVM Community Toolkit to simplify creation of ViewModels. Learn more about it here:

<https://learn.microsoft.com/dotnet/communitytoolkit/mvvm/generators/inotify>

We'll use the ViewModels namespace throughout the app, so add it to the GlobalUsings.cs file.

```
global using MauiStockTake.UI.ViewModels;
```

Creating the InputViewModel

So far, we've built the structure of the app using Shell, and we've built some authentication logic so that we can communicate with a REST API, but we haven't built any of the app's core functionality yet. Let's start doing that now.

View First vs ViewModel First

I usually build my apps with a ViewModel-first approach, rather than a View-first approach. This is because I am a .NET developer, not a UX or UI designer, so I tend to think of the apps I work on from the perspective of functional design rather than visual design. And, in all honesty, my visual design skills leave a lot to be desired, so I find it removes some of the cognitive load when I get around to the visual design if all the functional elements are already in place.

If you're more visually oriented, you might find it easier to build the View first, and then the ViewModel. You might also find that some frameworks and libraries either directly require or at least naturally lean toward one approach or the other.

Of course, if you have a UI designer on your team you may already have the visual design before you write a single line of code, in which case it might be easier to build out the UI first.

But one of the best benefits of the MVVM pattern is that you can do both at the same time. By using MVVM, one developer can work on the Views, and another can work on the ViewModels, without stepping on each other's toes, and you can wire them up once they're ready.

In the ViewModels folder, create a class called `InputViewModel` that inherits `BaseViewModel`. From our functional requirements, and from figure 9.10, we know that the `InputPage` will need four main controls:

- A `SearchBar` for looking up products
- A `CollectionView` for selecting the desired product from the search results
- A Stepper to indicate the number of units of the selected product that were counted
- A Button for recording the stock count

We haven't built the UI yet but given that we know it will need these controls, we can start adding some properties to the ViewModel to support them.

- The `SearchBar` will need a command to execute the search, so add an `ICommand` property and call it `SearchProductsCommand`.
- The `CollectionView` will need a collection to bind to for the search results, so add an `ObservableCollection` of type `ProductDto`. The `CollectionView` will also need a `ProductDto` to bind the selected item to, so add a `ProductDto` property called `SelectedProduct`.
- The `Stepper` will need an `int` to bind to, so add an `int` property called `Count`.
- The `Button` will need a command to save the stock count, so add an `ICommand` property and call it `AddCountCommand`.

If you've added all these properties your `InputViewModel` should look like listing 9.9.

Listing 9.9 InputViewModel.cs

```
using System.Collections.ObjectModel;#A
using System.Windows.Input;#B

namespace MauiStockTake.UI.ViewModels;

public class InputViewModel : BaseViewModel#C
{
    public ICommand SearchProductsCommand { get; set; }#D
```

```
public ICommand AddCountCommand { get; set; }#E  
public ObservableCollection<ProductDto> SearchResults { get;  
public ProductDto SelectedProduct { get; set; }#G  
public int Count { get; set; }#H  
}
```

This completes the definition of the `InputViewModel`. It doesn't do anything yet, we still need to add methods for the commands to execute, but the 'interface' (the bindable properties) for the View is complete. We'll come back and flesh out the functionality soon, but before we can use this `ViewModel` in a View, we need to register it with the service collection in `MauiProgram` so that we can inject it into the View's constructor. Add this code after the line that registers the `LoginPage`:

```
builder.Services.AddTransient<InputViewModel>();
```

9.5.3 The View

The corresponding View for the inventory workflow is the `InputPage`. We've already added the `InputPage` as a placeholder, but we need to add the layouts and controls to support this workflow.

Creating the InputPage

The `InputPage` provides the core workflow of our app and provides the functionality that will enable Mildred and her team to dispense with their inefficient paper and pen stock takes. At this point we've got a good understanding of what the page will look like and what it will do, so let's get started.

Let's get everything set up first in the code behind file. Open `InputPage.xaml.cs` and make the following changes:

1. Add a private read only field of type `InputViewModel`, called `_viewModel`

2. Inject the `IInputViewModel` into the page constructor
3. Assign the injected `ViewModel` to the field
4. Assign the page's `Navigation` property to the field's `Navigation` property
5. Set the page's binding context to the field
6. Remove the `Products` namespace

Once you have done this, your `InputPage.xaml.cs` file should look like listing 9.10.

Listing 9.10 InputPage.xaml.cs

```
namespace MauiStockTake.UI.Pages;

public partial class InputPage : ContentPage
{
    private readonly IInputViewModel _viewModel;#A

    public InputPage(IInputViewModel viewModel);#B
    {
        InitializeComponent();
        _viewModel = viewModel;#C
        _viewModel.Navigation = Navigation; #D
        BindingContext = _viewModel;#E
    }
}
```

Now we're ready to start building the UI. Open the `InputPage.xaml` file, and delete the `VerticalStackLayout` and its contents, leaving only the `ContentPage` tags. Next, we'll add the following to our page:

- A Grid for layout
- A SearchBar where the user can enter a search term and search for matching products
- A CollectionView to display the list of `SearchResults`
- An ActivityIndicator to show that search results are loading
- A Label to *show* the number of items the user has counted
- A Stepper to *record* the number of items the user has counted
- A Button that the user can tap to indicate that they want to save the count

Let's keep things simple and start with the layout. Add a Grid with Padding of thickness 20 on all sides and five rows. The figure below shows what these rows will hold and how they should be defined.

Figure 9.11 The relative row height, and contents, for the Grid that will make up the InputPage. Not shown is the ActivityIndicator, which will be in row 0 and have a rowspan of 5. This will make it appear in the center of the Grid when it is visible.

Row	Control	Relative Height	
0	SearchBar	1*	
1	Results Collection	3*	
2	Count Label	2*	4
3	Count Stepper	2*	
4	Submit button	1*	

Listing 9.11 shows InputPage.xaml looks like with this added Grid.

Listing 9.11 InputPage.xaml with the layout added

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiStockTake.Client.Pages.InputPage"
        Title="Input Page">
    <Grid Padding="20"#A
        RowDefinitions="*, 3*, 2*, 2*, *">#B
    </Grid>#C
</ContentPage>

```

Now that we've got our Page layout, we can start adding some controls. We'll start with the `SearchBar`, which will need its `SearchCommand` property bound to the `Command` we created in the `ViewModel`, and a `CommandParameter` to pass through the search term. Listing 9.12 shows the XAML code for the `SearchBar` control; add this inside the `Grid` on the `InputPage`.

Listing 9.12 The `SearchBar` control

```

<SearchBar x:Name="ProductSearchBar"#A
    Grid.Row="0"#B
    SearchCommand="{Binding SearchProductsCommand}"#C
    SearchCommandParameter="{Binding Text, Source={x:Refer
    Placeholder="Search for a product..."/>"#E

```

Next add the `CollectionView` to row 1 that will display the search results. We added an `ObservableCollection` of type `ProductDto` to the `InputViewModel`, so we can bind to this for the `CollectionView`'s `ItemsSource`. We also added a `ProductDto` called `SelectedProduct`, so we can bind the `CollectionView`'s `SelectedItem` property to this. We will also want to add a `DataTemplate` to display the product name and manufacturer name for each item.

The XAML code for the `CollectionView` is shown in listing 9.13.

Listing 9.13 The `CollectionView` control

```

<CollectionView Grid.Row="1"#A
    ItemsSource="{Binding SearchResults}"#B
    SelectedItem="{Binding SelectedProduct}"#C
   SelectionMode="Single"#D

```

```

        Margin="20, 0">
<CollectionView.ItemsLayout>#E
    <LinearItemsLayout ItemSpacing="10" #F
        Orientation="Vertical"/> #G
</CollectionView.ItemsLayout>
<CollectionView.ItemTemplate>
    <DataTemplate>
        <VerticalStackLayout>#H
            <Label Text="{Binding Name}"#I
                TextColor="Black"
                FontSize="Large"/>
            <Label Text="{Binding ManufacturerName}"#J
                TextColor="Gray"/>
        </VerticalStackLayout>
    </DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>

```

This `CollectionView` gives us not only a nice way to display the list of search results, but also a way for the user to select one to indicate which is the product they were looking for. The `CollectionView` doesn't include a loading indicator, so let's add an `ActivityIndicator` to the page so that the user knows something is happening once they've submitted their search.

We'll add it to the same row of the `Grid` as the `CollectionView`, which will help the user to understand that it's the results collection that's loading when the indicator shows. We'll bind the `IsVisible` property to the `IsLoading` property of the `ViewModel`, which we can set based on when we're loading data.

[Listing 9.14](#) shows the XAML code for the `ActivityIndicator`. Add this to the `InputPage` after the `CollectionView`.

Listing 9.14 The `ActivityIndicator` control

```

<ActivityIndicator Grid.Row="1"#A
    HorizontalOptions="CenterAndExpand"#B
    VerticalOptions="CenterAndExpand" #C
    IsRunning="True"#D
    IsVisible="{Binding IsLoading}"/> #E

```

There are two parts to recording a stock count; the product being recorded,

and the number of items counted. We've added the product part, now let's move on to the count. We'll add a Label to show the current count, and a Stepper so that the user can increase or decrease the count. Listing 9.15 shows the XAML code for these controls.

Listing 9.15 The Count Label and Stepper controls

```
<Label Text="{Binding Count}"#A  
    FontSize="Header" #B  
    HorizontalOptions="Center"#C  
    VerticalOptions="Center"#C  
    Grid.Row="2"/>#D  
  
<Stepper HorizontalOptions="Center"#E  
    VerticalOptions="Center"#E  
    Value="{Binding Count}"#F  
    Grid.Row="3"/>#G
```

Our InputPage is almost finished. The last step is to add a Button so that the user can indicate that their stock count is ready to be recorded. Listing 9.16 shows the XAML code for the Button control.

Listing 9.16 The Button control

```
<Button Text="Add count"#A  
    Command="{Binding AddCountCommand}" #B  
    Grid.Row="4"/>#C
```

This completes the code for the InputPage. Now you can run the app, and you should see something like figure 9.12.

Figure 9.12 The InputPage of the MauiStockTake app, seen here running on Android



Finally, we can see our running app! All the controls we see here in the UI are bound to the ViewModel. But, while the ViewModel exposes some properties for the UI to bind to, it doesn't provide any functionality. For that, we need to wire up the services, which we will do next.

9.5.4 Adding the Search functionality

The ProductService will be used by the InputViewModel to look up products based on a search term entered by the user. Before we build the ProductService, we will define what functionality the ViewModel needs from it.

Open `InputViewModel.cs` and inject the `IProductService` into the constructor. To access it outside the constructor, assign it to a private readonly field. The constructor is shown in listing 9.17.

Listing 9.17 The updated InputViewModel constructor and field

```
private readonly IProductService _productService;  
  
public InputViewModel(IProductService productService)  
{  
    _productService = productService;  
}
```

Add a reference to `MauiStockTake.Client.Services` to `GlobalUsings.cs` to import the namespace for this service.

Next, add a new private method of type `Task` called `UpdateSearchResults`, that takes a parameter of type `string` called `searchTerm`). Make the method `async` so that it doesn't block the UI while search results are being retrieved from the API.

Inside the method, we want to do the following steps:

1. Set the `IsLoading` property (inherited from the `BaseViewModel`) to true. We'll bind the `ActivityIndicator` in the UI to this property.
2. Clear the `SearchResults` `ObservableCollection` so that only results for the current search are shown. As `ObservableCollections` automatically notify the UI of changes to their content, we clear the collection rather than creating a new instance.
3. Call the `SearchProducts` method we just defined and assign the result to a temporary variable.
4. Set `IsLoading` back to false to hide the `ActivityIndicator`.

5. Iterate through the list of search results, and add each one to the `ObservableCollection`.

The code for this method is shown in listing 9.18.

Listing 9.18 The `UpdateSearchResults` method of the `InputViewModel`

```
private async Task UpdateSearchResults(string searchTerm) #A
{
    IsLoading = true; #B

    SearchResults.Clear(); #C

    var results = await _productService.SearchProducts(searchTerm)

    IsLoading = false; #E

    results.ForEach(res => SearchResults.Add(res)); #F
}
```

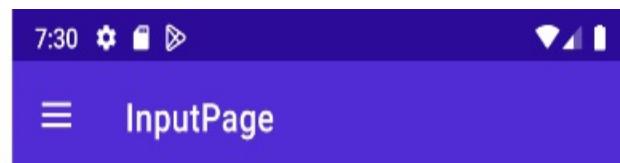
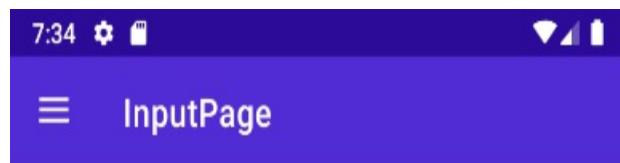
The last step to make this work is to wire up the `SearchProductsCommand` to use this method. Add the following line to the `InputViewModel` constructor.

```
SearchProductsCommand = new Command<string>(async (term) => await
```

Note that the `Command` here is typed to `string`. This is what allows the `Command` binding to send a `CommandParameter`.

Run the app and try searching for a product. You should see an `ActivityIndicator` spinning while the app is loading results from the API, and if you've entered a term that matches a product in the database, you'll see some results (appendix D contains the full list of products in the `MauiStockTake` database), similar to the following figure.

Figure 9.13 MauiStockTake searching for and then displaying products, seen here running on Android



9.5.5 Adding the Inventory Functionality

Now that we've successfully got our app searching for products, we need a way to record stock count results. If you've explored the app a little you might notice that the Stepper buttons (+ and -) don't seem to do anything. This is because both the Stepper and the Label above it are bound to the Count property, but the Count property doesn't notify the UI of changes. The value is being set when we tap the + and – buttons, but that change is not reflected in the UI.

We can fix this by updating the Count property in the `InputViewModel` to use a backing field and raise a property changed notification when it gets updated. Update the Count property in your `InputViewModel` to match listing 9.19.

Listing 9.19 The `InputViewModel` Count property

```
private int _count = 0;#A

public int Count #B
{
    get => _count;#C
    set
    {
        _count = value;#D
        OnPropertyChanged();#E
    }
}
```

If you run the app again now, you should see the count indicator changing in response to the Stepper.

Now that we've got a product and a count, we need to record them. This functionality is provided by the `IInventoryService`, so let's inject this into the `InputViewModel` constructor, as in listing 9.20 (with changes in **bold**).

Listing 9.20 The `InputViewModel` constructor

```
private readonly IInventoryService _inventoryService;#A

public InputViewModel(IProductService productService, IInventor
```

```
{  
    _inventoryService = inventoryService;#C  
    ...  
}
```

Next let's add the method to add a stock count using the `IInventoryService`. This method will need to do the following things:

- Verify that the user has selected a product, and display an alert if they haven't
- Send the stock count to the API using the `IInventoryService`
- Display a message to the user indicating whether the action was successful

Listing 9.21 shows the `AddCount` method to be added to the `InputViewModel`.

Listing 9.21 The InputViewModel AddCount method

```
private async Task AddCount()#A  
{  
if (SelectedProduct is null)#B  
{  
    await App.Current.MainPage.DisplayAlert("Product Required",  
    return;  
}  
  
IsLoading = true;#C  
  
    var added = await _inventoryService.AddStockCount(SelectedPr  
  
IsLoading = false;#E  
  
    if (added)#F  
    {  
        await App.Current.MainPage.DisplayAlert("Added", "Stock  
    }  
    else  
    {  
        await App.Current.MainPage.DisplayAlert("Error", "Someth  
    }  
}
```

The last step is to assign the new method to the `Execute` property of the `AddCountCommand`. Add the following line to the constructor:

```
AddCountCommand = new Command(async () => await AddCount());
```

At this point, the stock take workflow is functionally complete. You can run the app, search for products, and record a stock count. There are two remaining UX bugs though, one of which is that the count only allows input via the Stepper, which could be tedious if the user wants to record a count in the 10s or hundreds (anything greater than 10 will be annoying to have to enter via the Stepper). We'll come back to this problem in chapter 11. The other, which we should tidy up now, is that the search results stick around after recording a stock count. Ideally the form will reset after recording data, ready for the next stock count.

There's not much to do to get this working. We'll need to make a small change to the `InputPage` XAML and a small change to the `InputViewModel`. Currently, the `SearchBar` control calls a `Command` on the `ViewModel` and sends its `Text` property as a `Command` parameter. Instead, we'll bind the `Text` property to a property on the `ViewModel` and use this to call the `SearchProducts` method on the `IProductService`. Then we'll simply set it to an empty string to clear the search term and clear the `ObservableCollection` of search results.

Let's start by updating binding properties, then we can clear the form. Add a `string` property called `SearchTerm` to the `InputViewModel`. Then, update the `UpdateSearchResults` method to use this rather than a parameter, and update the command definition in the constructor.

Next, add a method called `ResetForm`. In this method, we will:

1. Clear the `SearchResults` collection
2. Set the `Count` to 0
3. Set the `SelectedProduct` to `null`
4. Set the `SearchTerm` to an empty string
5. Raise a property changed notification for the `SearchTerm` property

Once we've added the method, we can call it from the `AddCount` method to reset the form once the count has been successfully recorded.

Listing 9.22 shows the full code for the updated `InputViewModel`, with

changes in **bold**.

Listing 9.22 The updated InputViewModel

```
public class InputViewModel : BaseViewModel
{
    ...
    public string SearchTerm { get; set; }#A

    public InputViewModel(IProductService productService, IInven
    {
        ...
        SearchProductsCommand = new Command(async () => await U
    }

    private async Task UpdateSearchResults()#C
    {
        IsLoading = true;

        SearchResults.Clear();

        var results = await _productService.SearchProducts(Search
        IsLoading = false;

        results.ForEach(res => SearchResults.Add(res));
    }

    private async Task AddCount()
    {
        ...

        if (added)
        {
            await App.Current.MainPage.DisplayAlert("Added", "St
            ResetForm();#E
        }
        else
        {
            ...
        }
    }

    private void ResetForm()#F
```

```

    {
        SearchResults.Clear();#G
        Count = 0;#H
        SelectedProduct = null;#I
        SearchTerm = string.Empty; #J
        OnPropertyChanged(nameof(SearchTerm));#K
    }
}

```

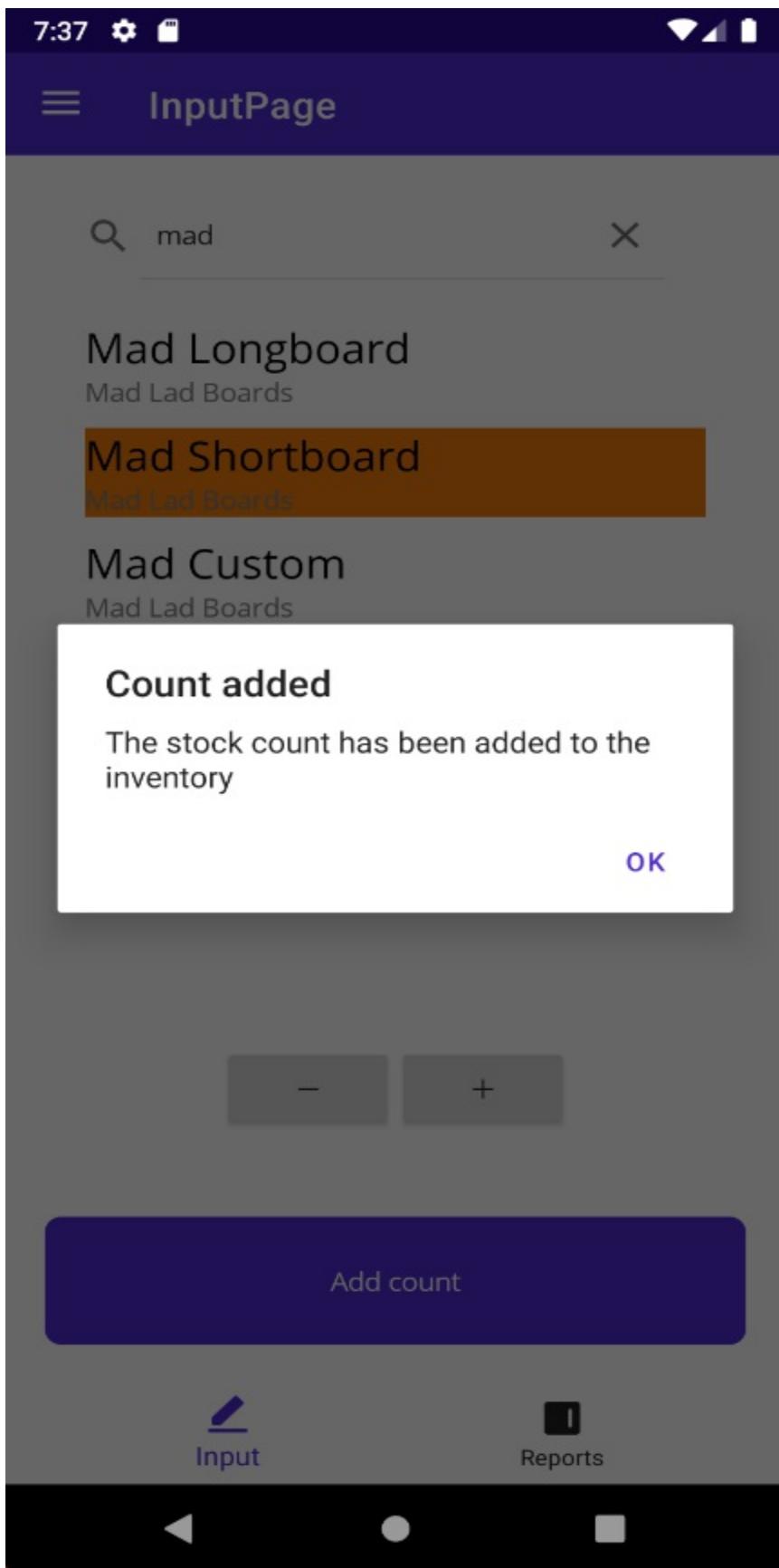
Now that we've updated the ViewModel to reset the form after successfully adding a stock count, we need to make a small change to the UI. The `SearchBar` control currently sends a command parameter, we need to remove this, and bind the `Text` property to the `SearchTerm` property on the `ViewModel`. Listing 9.23 shows the updated XAML code for the `SearchBar` control in the `InputPage`. The added binding is in **bold**, but note the removed command parameter is not shown (it has been replaced by the `Text` binding).

Listing 9.23 The updated `SearchBar` control

```
<SearchBar x:Name="searchBar"
    Grid.Row="0"
    TextColor="{StaticResource PrimaryColor}"
    SearchCommand="{Binding SearchProductsCommand}"
    Text="{Binding SearchTerm}"
    Placeholder="Search for a product..."/>
```

That completes everything we need for the stock take workflow (except, perhaps, improving the UX for entering large numbers). Go ahead and run the app, and make sure you can search for products and record stock counts. You should see something like figure 9.14.

Figure 9.14 A successful stock count recording in the MauiStockTake app, seen here running on Android



9.6 Review of the MauiStockTake App So Far

Congratulations on building your first functional real-world .NET MAUI app! Up until this chapter, all the examples we've been building have been simple demos. MauiTodo is something of an exception, although it's still very simplistic. MauiStockTake is our first major project.

We've built the MauiStockTake app using the MVVM pattern. Using this approach, we have a clean separation between our UI logic, presentation logic and business logic. Using the `InputPage` and the stock take workflow as an example, we have UI logic in the `InputPage`, presentation logic in the `InputViewModel`, and business logic in the models (`ProductDto` and `StockCountDto`), and services (`IInventoryService`, `IProductService`, `IAuthService` and their implementations).

Each of these pieces have been built independently and can be maintained independently. We could completely change the look of the UI without having to rebuild any other parts of the app. Likewise, we could switch out the REST API for GraphQL and all we would have to do is update the functionality inside the service implementations. No other parts of the app, including the interfaces, would have to change.

We've followed the principle of dependency inversion (defining dependencies where we intend to consume them), starting with the UI. The `InputViewModel` is designed to expose functionality and state required by the `InputPage`. I've cheated a little in that I've provided you with interfaces that provide functionality required by the `InputViewModel`; but this is to simplify things. I actually built the `InputViewModel` first, and defined the interfaces as I was building it, then built the implementations afterwards.

The MVVM pattern isn't the only way to build apps with .NET MAUI, but it is the most popular. And for good reason – MVVM helps us build maintainable apps. The MauiStockTake app is architected for maintenance, making it simple to grow and change.

There's still more functionality to add, and there are a few things we can do

to improve our app. We'll keep building on MauiStockTake over the next chapters, but before we move on, treat yourself to a slice of cake. You've earned it.

9.7 Summary

- We use the MVVM pattern in .NET MAUI to improve separation of concerns. Presentation logic goes in the View, business logic goes in the ViewModel, domain logic goes in the Model.
- By separating these concerns, we have code that is more maintainable and more testable.
- We can use Command binding to execute functionality in ViewModels from Views.
- `IPropertyChanged` defines events that tell Views to update their content. We implement this interface on ViewModels to raise this event and update the UI when state in the ViewModel changes.
- Use the dependency inversion principle to define functionality where you intend to consume it, rather than where it's provided, starting with your View.
- Use dependency injection and the `ServiceCollection` in the host builder to manage dependencies throughout your app.
- You can extend any controls with behaviors, by adding functionality to the control without having to subclass it.
- The .NET MAUI Community Toolkit contains a plethora of useful features many developers will utilise in their apps.
- `EventToCommandBehavior` lets us respond to events with proprietary parameters.

10 Styles, themes, and multi-platform layouts

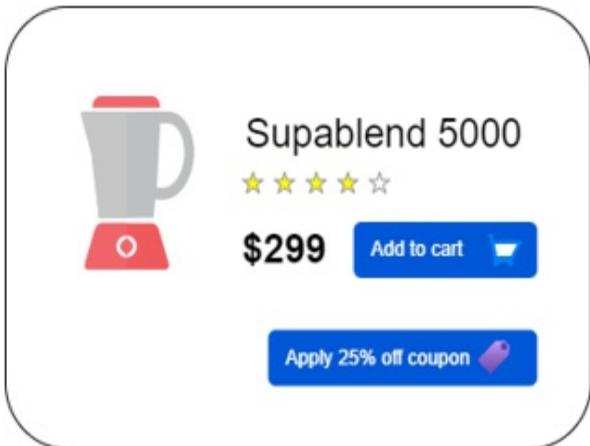
This chapter covers

- Styling controls
- App themes and light/dark mode
- Triggers and visual state
- Features of different device paradigms

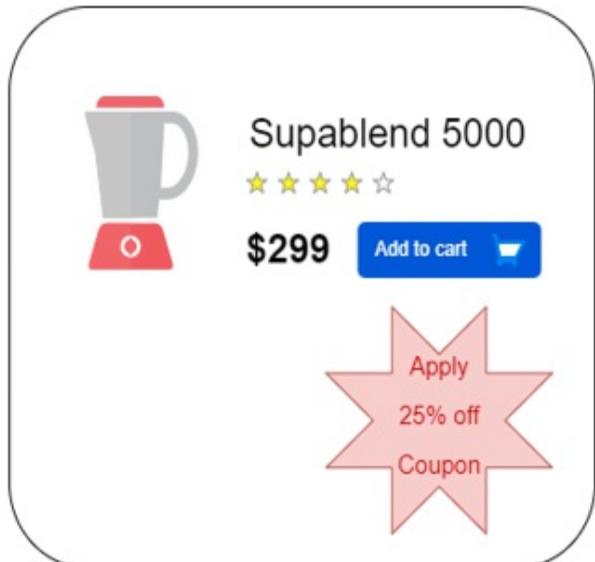
Back in chapter 1 we looked at the importance of maintaining a consistent UX throughout your app, and across different platforms, and we touched on it again in chapter 5. Whether you strive for the “pixel perfect” approach, or just want your app to feel like your app, wherever it is used, keeping your UI consistent is critical and helps users understand how your app works.

For example, using a primary and secondary color helps your users to understand the purpose of a button, as does using specific button shapes (for example, it’s common to use a circular FAB to indicate an additive action). On the other hand, using too many different shaped or colored buttons could lead to confusion. Let’s see an extreme example in figure 10.1.

Figure 10.1 Two versions of an app UI. In both cases, a button is presented which the user can tap to apply a 25% discount to their order. In the example on the left, the button looks like all the other buttons in the app, and the user can easily tell what to do. On the right, the button looks completely different, and it is not easy to tell that the star is a button that can be tapped.



Buttons are consistent and the user can easily tell what to do



Buttons are not consistent and the user can't easily tell that the star is a button

With the example on the left, while having two identical buttons dilutes the call to action (CTA), and the coupon button should probably be in a secondary style, it's easy to see that there is a button you can tap to apply a 25% discount to your order. On the right, while the star may seem more visually striking, it's not consistent with the app's design language and gives no indication to the user that it can be tapped to apply the discount.

This example is extreme, but even small variations in the presentation of controls can cause confusion. In this chapter, we'll see how you can use Styles to ensure that the look and feel of your controls is the same throughout your app, and across different platforms (even if they are not identical). We'll also see how you can use themes; the most common themes are light and dark, and users often like to choose between light and dark modes, or let the app synchronize with the light or dark mode of the OS.

Maintaining consistency across device types and OSes is important, but it's also important to leverage the UX expectations users have for different paradigms. We'll see in this chapter how you can take advantage of UX features users have come to expect on desktop and mobile devices.

10.1 Creating a consistent look and feel

We've made some small changes to the way controls look in our .NET MAUI apps so far, but we haven't changed the overall *style* of an app. We want MauiStockTake to reflect the surfing look and feel of Mildred's business, so let's update the app's styles to match.

In Chapter 1 (and again in Chapter 5) we looked at the *consistent* but not *identical* approach to UI, and this is achieved in .NET MAUI using styles. Let's see how we can use styles to apply this to MauiStockTake.

10.1.1 Styles

All controls in .NET MAUI have properties that can be customized to change their appearance. Some of these, like `TextColor` on the `Label` and `Entry` controls, are properties of the control itself (although, as you can see, some controls have properties with the same names). Others, like `HorizontalOptions` and `WidthRequest` are inherited from the `View` and `VisualElement` base classes.

You can modify any of these properties on any control in your app. But applying these modifications each time you add a control is not only laborious, but it also introduces the risk of human error, which could undermine the consistency we're striving for. To avoid this problem, we can use styles.

Styles are collections customizations for a specific control type. The .NET MAUI templates we have been using have some styles already built in, which is why, for example, the `Button` looks roughly the same on each platform.

Let's see how these default app-wide styles are defined, and how we can use them to customize the MauiStockTake app.

Styling Buttons

In both the default template and the `blankmaui` template, a set of styles have been defined (we'll look at implicit vs explicit styles in a few pages). If you

open `App.xaml`, you can see that two additional files are loaded into the app's resource dictionary. Both are in the `Resources` folder; one is called `Colors.xaml`, and the other is called `Styles.xaml`. These files contain a collection of colors used throughout the app and a collection of default styles (which in turn uses the colors defined in `Colors.xaml`) respectively. As they are loaded into the app's resource dictionary, they are applied across the whole of the app.

Let's start by updating the colors. In `Colors.xaml`, update the `Primary` and `Secondary` values to match those in listing 10.1, and add the additional `PrimaryBackground` and `SecondaryBackground` colour definitions.

Listing 10.1 Colour definitions to add to the `Colors.xaml` resource dictionary

```
<Color x:Key="PrimaryColor">#215377</Color>
<Color x:Key="SecondaryColor">#8dacb9</Color>
<Color x:Key="PrimaryBackground">#8dacb9</Color>
<Color x:Key="SecondaryBackground">#b4bcc7</Color>
```

If you run the app now, you'll notice the change in `Button` color right away. The two background colors we've added are for styling the `Page` which we'll get to shortly, but let's take a look at how those `Button` colors are applied.

Open the `Styles.xaml` file and look for the style with a `TargetType` of `Button`. Styles must define a `TargetType` and can then use `Setters` to define how properties of that target type should look. For this style with a `TargetType` of `Button`, we can use a `Setter` for any property of a `Button` that we want to control.

For the `Button` style, there is a `Setter` for the `BackgroundColor` property. This is using `AppThemeBinding`, which we'll look at in the next section, but we can see in this `Setter` that when the app is running in light mode, the `BackgroundColor` property of a `Button` will be set to the `Primary` color defined in the `Colors.xaml` file, and when the app is running in dark mode, it will be white. You can see the setter for this property here:

```
<Setter Property="BackgroundColor" Value="{AppThemeBinding Light=
```

There are two more changes we want to make to the `Button` style to match

Mildred's brand guidelines. The first is that we want to add a setter for the HeightRequest property and make it 50, so that every button in the app is the same height. The second is that we want to change the CornerRadius setter from 8 to 25. This is half of the height and will give us perfectly rounded edges on every Button.

Add the Setter for HeightRequest now. Depending on your IDE or code editor, you might notice that Intellisense gives you the list of properties. Then change the Value in the Setter for CornerRadius. Finally, add another Setter for MinimumWidthRequest and set the Value as 100. If you run the app now, you'll see the updated Button style on the LoginPage, as in figure 10.2.

Figure 10.2 The Login button on the LoginPage is now styled and meets the branding guidelines for this app. Even though we haven't customised the button directly, the app's styles have been applied to give us the visual style we wanted.



Styling Pages

Pages aren't as customizable as other views, but they do have some properties that you might like to make consistent across your app. One of these might be Padding, but you may also want to tailor this to different platforms and device types.

For MauiStockTake, we're going to customize the page background to give us a gradient that's aligned with the beachy, surfing theme of Mildred's brand. In Styles.xaml, look for the style with a TargetType of Page. There is already a Setter here for the BackgroundColor property; we're going to delete this and replace it with a Setter for the Background property.

What is the difference between Background and BackgroundColor?

The VisualElement base class in .NET MAUI, from which all views are derived, has both a Background and a BackgroundColor property. They are similar in that they both set the background of a view, but differ in that BackgroundColor is of type Color, and Background is of type Brush.

Brushes allow you to 'paint' an area (such as the background of a view); but in addition to allowing just a single color (with the SolidColorBrush, which in the case of Background is identical in function to setting the BackgroundColor property), you also have a LinearGradientBrush and a RadialGradientBrush.

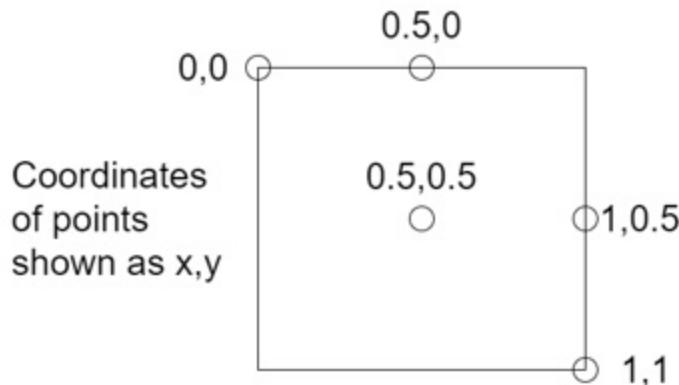
We've mentioned brushes before and we're going to use them here to set a gradient as the Background property of a Page. If you just want a single color, you could achieve this by setting the Background property and using a SolidColorBrush, but a simpler approach is to set the Background property.

We'll use gradient brushes again in the next chapter.

Because Background is of type Brush, and we want to assign a gradient to it, we won't be able to use the inline Value property in the Setter. Instead, we'll add it explicitly using tags. Using this approach, we can assign a LinearGradientBrush to the Value property and specify its StartPoint and EndPoint properties. These are of type Point and are made up of an x and y value, each of which is between 0 and 1 and represents a proportion of the view's width and height respectively.

Starting at the top-left with $0, 0$, a value of $0.5, 0.5$ would be the center of the view, and $1, 1$ would be the bottom right, as you can see in figure 10.3.

Figure 10.3 Points in a view. Each point is defined by an x and y coordinate expressed as a fraction (between 0 and 1) of the width (for x) or height (for y) of the view.



Our Page gradient is going to run from bottom left to top right (it's quite a subtle gradient so we could change these without too much impact), so we'll add a `StartPoint` of $1, 0$ and an `EndPoint` of $0, 1$.

The `StartPoint` and `Endpoint` define the *axis* of the gradient – the line from start to finish along which the gradient will run. In addition to the axis, we also need to define *gradient stops*, which specify colors at positions along the axis. `GradientStop` has two properties: `color` and `offset`. `color` is the color to assign to that position, and `offset` is a fractional position along the axis, ranging from 0 to 1.

You can specify as many gradient stops as you like, but for MauiStockTake we are going to just provide two, using the `PrimaryBackground` and `SecondaryBackground` values we specified in the `colors.xaml` file. We'll add the `PrimaryBackground` to the start of the axis by giving it a value of 0, and place the `SecondaryBackground` at the end of the axis by giving it a value of 1.

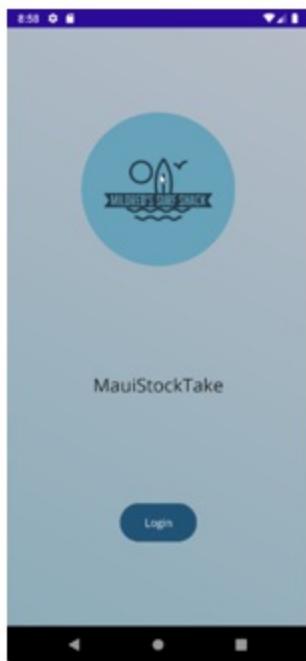
Listing 10.2 shows the Setter to add to the style with a `TargetType` of `Page` in the `Styles.xaml` file. Delete the Setter for the `Background` property and add the Setter from this listing.

Listing 10.2 The Background Setter for the Page style

```
<Setter Property="Background">
    <Setter.Value>
        <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
            <GradientStop Color="{StaticResource PrimaryBackground}" StopOffset="0.0" />
            <GradientStop Color="{StaticResource SecondaryBackground}" StopOffset="1.0" />
        </LinearGradientBrush>
    </Setter.Value>
</Setter>
```

Run the app now, and you should see the LoginPage automatically inheriting the Background property we've defined in this style, as in figure 10.4.

Figure 10.4 The LoginPage has a LinearGradient as its Background property. This gradient hasn't been set on this page, but has been applied to a style that targets the Page type and is imported at the app level.

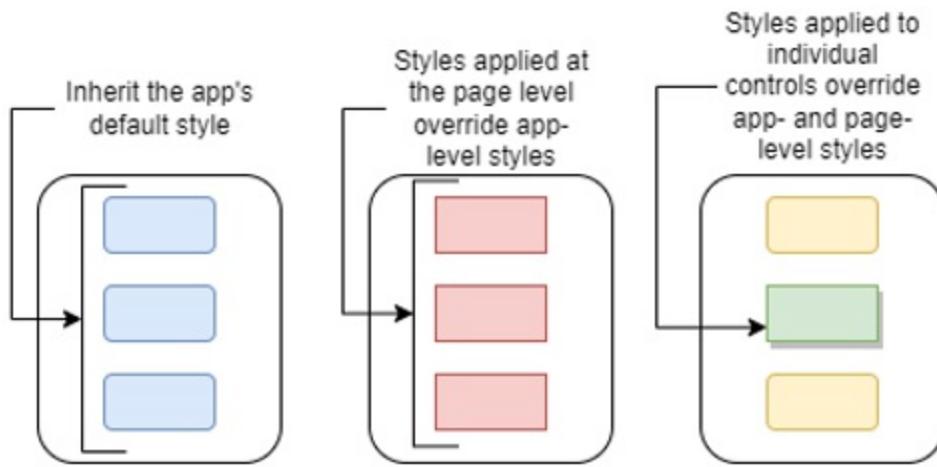


Style Hierarchy

Styles are added to a resource dictionary, which can either be the app's resource dictionary (which is the case with the built-in styles) or a page's resource dictionary. Control properties are applied hierarchically, with Styles defined in the app's resource dictionary applied first, then those in the page's

resource dictionary, and finally any properties explicitly defined on the control. Figure 10.5 illustrates this hierarchy.

Figure 10.5 Control property values are applied hierarchically, with app level-level styles being applied by default, page-level styles overriding app-level styles, and values set directly on controls overriding any styles.



You can see from figure 10.xx that if you apply a style at the app level, it will apply to all matching controls in the app. If you apply a style at the page level, it will apply to all matching controls on that page and will override app-level styles. Properties defined on a control override any app or page styles.

Styles that are narrower in scope (i.e., defined at a more granular level) take precedence over broadly scoped styles. In addition to the scope of the style, an explicit style will take preference over an implicit style.

Implicit vs Explicit Styles

The styles that we've looked at so far are *implicit* styles. This means that they are defined against a target type, and every view of that type that is hierarchically in scope of the style will apply it.

You can also use *explicit* styles. An explicit style is defined almost identically to an implicit style, except that it adds a *key*. A key is an identifier by which a resource can be referred to. A view will not apply an explicit style unless the

style is explicitly assigned to it using, unsurprisingly, the `Style` property.

The count label is a little bland at the moment, and also not particularly easy to read. Let's use an explicit style to improve it.

NOTE

In a real-world app, it would make much more sense to apply these properties directly to the view. The value gained from creating a style is when it applies to more than one control.

In the `Styles.xaml` file, let's create a new explicit style that targets the `Label` type, and give it a key of `CountLabelStyle`. The count label already has some properties defined (`FontSize`, `HorizontalOptions` and `VerticalOptions`), so we can start by adding these to our style. Lastly, we can specify the `TextColor` property and set it as the app's Primary color. Listing 10.3 shows the style to add to the `Styles.xaml` file.

Listing 10.3 The `CountLabelStyle` style

```
<Style TargetType="Label" x:Key="CountLabelStyle">
    <Setter Property="FontSize" Value="64"/>
    <Setter Property="HorizontalOptions" Value="Center"/>
    <Setter Property="VerticalOptions" Value="Center"/>
    <Setter Property="TextColor" Value="{StaticResource Primary}" />
</Style>
```

Once you've added this style, we can remove the properties defined here from the count label and assign the style instead.

```
<Label Grid.Row="2"
       Text="{Binding Count}"
       Style="{StaticResource CountLabelStyle}"/>
```

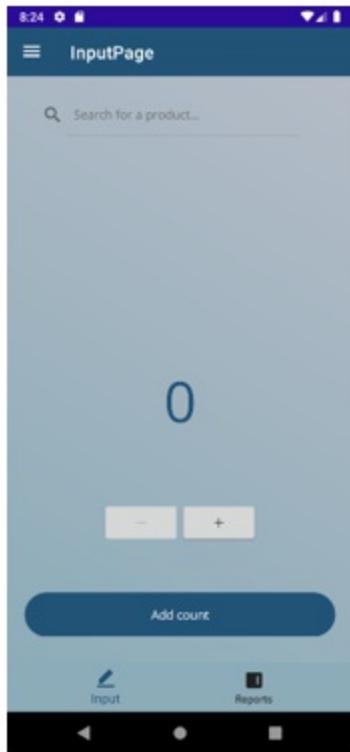
Once you've made these changes, run the app, and it should look something like figure 10.6.

Figure 10.6 An explicit style is applied to the count label to control its horizontal and vertical position, the font size, and the text color. These customisations can be applied to any view with a matching target type (in this case `Label`) by using the `Style` property and referencing this explicit style.



One thing that still doesn't fit in here is the tab bar. In figure 10.xx you can see that it has a white background that doesn't fit in with the rest of the app, so let's update the style for Shell to make this match. In `Styles.xaml`, find the style for `Shell`, and the `Setter` for `Shell.TabBarBackgroundColor`. Delete the value and replace it with `{StaticResource PrimaryBackground}`. Run the app again, and you should see the tab bar blend into the page background, as in figure 10.7.

Figure 10.7 The tab bar background in MauiStockTake is now consistent with the rest of the app, and is controlled by a Setter in the app's Styles



10.1.2 Themes

As developers, we're used to being able to customize the theme or appearance of apps we use. Visual Studio Code, by far the world's most popular developer tool (according to the Stack Overflow developer survey, see: <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-new-collab-tools>), allows extensive customization and theming, with themes being amongst the most downloaded extensions.

Customizable color themes or palettes are common in many apps, but even those that don't offer them nearly always offer an option to switch between light mode and dark mode. I conducted a Twitter poll to help understand how important this feature is, and while this is by no means a scientific study, the results, seen in figure 10.8, provide a clear indication that light and dark mode support is generally considered to be important.

Figure 10.8 A short Twitter poll asking for opinions about color themes in apps. The responses overwhelmingly indicate that providing a light and dark mode should be a priority.



In this section, we'll see how you can provide support for light and dark mode in your app, and how you can respond to the system theme too. We'll also look at how you can offer a range of color themes for users to choose from.

Light Mode and Dark Mode

Mildred and her team will often be taking inventory at night, so it's a good idea to add dark mode support to MauiStockTake so that they don't get unnecessary eye strain. Fortunately, support for light and dark mode comes built-in to the templates in .NET MAUI. Without changing anything, if you create a new project using either the default or `blankmaui` templates, or open one of the projects from earlier chapters, you'll notice that the colors in the app change when you change your device from dark mode to light mode, or vice versa.

Let's update the dark mode colors for MauiStockTake. Open `Colors.xaml` and add the color definitions from listing 10.4.

Listing 10.4 Dark mode color definitions for MauiStockTake

```
<Color x:Key="PrimaryDark">#7b98aa</Color>
<Color x:Key="PrimaryDarkBackground">#141d31</Color>
<Color x:Key="SecondaryDarkBackground">#212f51</Color>
```

Now that we've got some dark mode colors defined, we can update the app's styles to use these colors. You may have spotted some of the styles using a

markup extension called `AppThemeBinding`. `AppThemeBinding` lets you respond to the system's light and dark modes, specifying a different value for each mode.

If you look through `Styles.xaml`, you'll see lots of examples of this, and you may spot plenty of opportunities for customizing the light and dark mode behavior of your apps. For MauiStockTake, we'll use `AppThemeBinding` to control the light mode and dark mode of the following:

- Page background
- Activity indicator
- Button
- The flyout background
- The tab bar
- The navigation (title) bar

Start with the style for `ActivityIndicator`; for light mode, it currently uses the `Primary` color, but for dark mode uses `White`. **Change the dark mode color** to the `PrimaryDark` color we've defined.

We can do the same thing in the style for `Button` with the `Setter` for the `Background` property. For the `TextColor` `Setter`, **I remove the `AppThemeBinding` and just set the value to the `White` `StaticResource`**, as the `Button` `Background` is no longer white in dark mode, and white text will contrast just as well with `PrimaryDark` as with `Primary`.

The `Setter` for the `Background` property that we added to the `Page` style has two gradient stops, each one referring to a color in the resource dictionary. It's easy enough to change these to use `AppThemeBinding`; let's do this, but let's also take it a step further by making the whole gradient brush a resource.

Cut the whole `LinearGradientBrush` from between the `<Setter.Value>` tags and paste it into the `Colors.xaml` file. You'll see that there are already some `SolidColorBrush` resources defined here; for our gradient, once we've pasted it all we need to do is add a key like these other brushes. **Give it a key of `BackgroundGradient`, then update the gradient stops to use `AppThemeBinding`.** Keep the existing colors for light mode, and add the new dark mode background colors we added in listing 10.4. Your new gradient

brush resource should look like listing 10.5

Listing 10.5 The BackgroundGradient resource

```
<LinearGradientBrush x:Key="BackgroundGradient" StartPoint="0,1"
    <GradientStop Color="{AppThemeBinding Light={StaticResource P
        <GradientStop Color="{AppThemeBinding Light={StaticResource S
    </LinearGradientBrush>                                #B
```

Now that the gradient background is a defined resource, we can simplify the Setter for the Background property of the Page style. **Make the <Setter ...> a single line self-closing tag, and set the value to the BackgroundGradient StaticResource.** The Setter should now look like this:

```
<Setter Property="Background" Value="{StaticResource BackgroundGr
```

Next let's fix the tab bar. In the previous section, we removed the AppThemeBinding for Shell.TabBarBackgroundColor and replaced it with the PrimaryBackground static resource. **Let's add the AppThemeBinding back in, and use PrimaryBackground for light mode, and PrimaryDarkBackground for dark mode.** The Setter for Shell.TabBarBackgroundColor should look like this:

```
<Setter Property="Shell.TabBarBackgroundColor" Value="{AppThemeBi
```

The flyout background also needs to be updated. This is a simple change; **add a Setter for Shell.FlyoutBackgroundColor and use AppThemeBinding to set PrimaryBackground as the light value and PrimaryDarkBackground for dark.** The Setter should look like this:

```
<Setter Property="Shell.FlyoutBackgroundColor" Value="{AppThemeBi
```

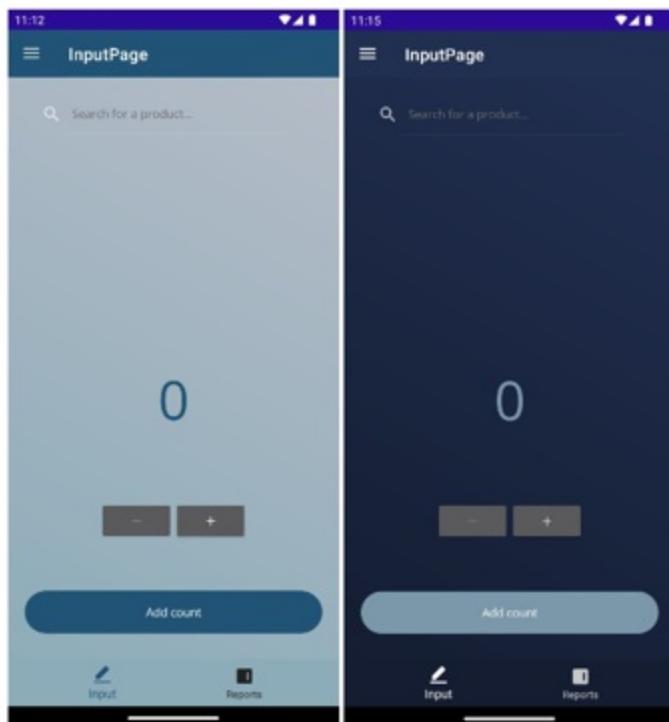
The last change to make for Shell is to update the Setter for the Background property. This value is used to set the background of the navigation bar that holds the menu button and title. It's already got AppThemeBinding; we can leave Primary as the value for light mode, but let's **change the dark mode value from Gray950 to SecondaryDarkBackground.** This will make both the top and bottom bars blend in with the background gradient. But as the gradient runs at a diagonal, a slight border will be visible, giving a nice UI

effect.

The final thing we should update is the explicit style we created for the CounterLabel. **Update the Setter for the TextColor property to use AppThemeBinding, keeping the Primary StaticResource for light mode and using PrimaryDark for dark mode.**

With all these changes made, your InputPage should look similar to figure 10.9.

Figure 10.9 MauiStockTake running light mode (left) and dark mode (right). Because AppThemeBinding is used, this mode is not selected within the app, but is bound to the light or dark theme of the OS.



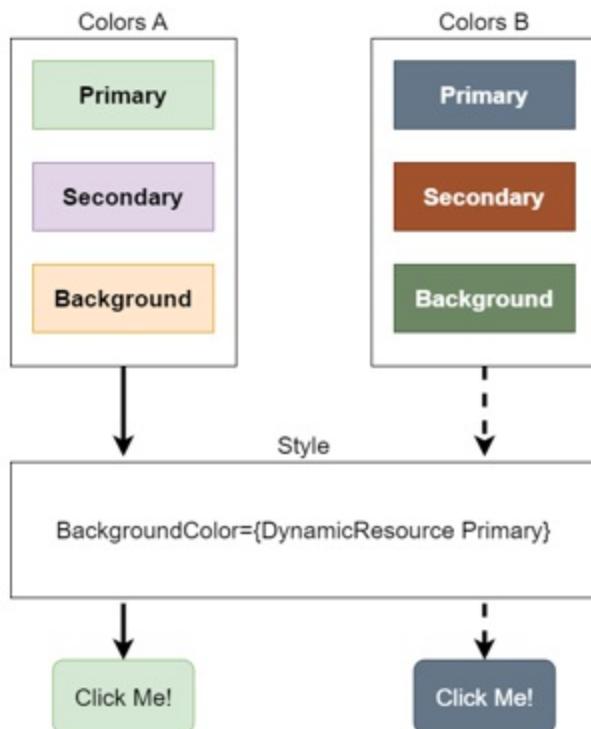
You can see in Styles.xaml there is scope to customise many more styles, but the AppThemeBinding markup extension can be used anywhere, not just for setting colors. For example, we could use a different version of the company logo on the login screen for dark mode if we wanted, or even have different values for labels.

Custom Themes

You can go beyond light and dark mode and offer full themes for your apps. Looking at the poll in figure 10.8, this doesn't seem as high demand a feature as light and dark mode support. Although it's worth noting that this is far from a rigorous scientific study, and was conducted on an app that does, in fact, offer customizable themes.

We've been using the `StaticResource` markup extension to refer to colors in the App's merged resource dictionary. As the name suggests, the value is considered static, so does not change at runtime. However, if you use its `DynamicResource` counterpart, the UI will change at runtime to reflect changes in the source value. With this approach, you can swap out the color palette at runtime, and using `DynamicResource`, your views will automatically update to reflect the new colors, as seen in figure 10.10.

Figure 10.10 This app contains two color palettes and one set of styles. The styles use `DynamicResource`, so the color palette can be swapped out at runtime, and views that apply the style will automatically update in real-time.

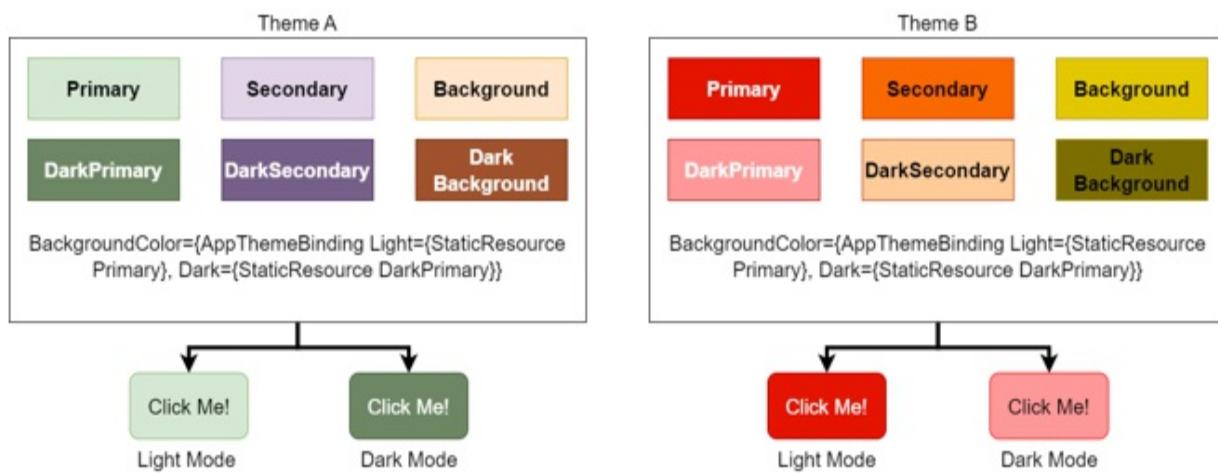


The .NET MAUI documentation covers this approach to theming apps, and you can read about it here: <https://learn.microsoft.com/dotnet/maui/user-interface/themes>

[interface/theming](#). There is a limitation though, which is that `AppThemeBinding` and `DynamicResource` don't work together. It will work well if you want to provide your own user-selectable color schemes (using just `DynamicResource`, as per the example in the documentation) but won't allow you to bind to OS light mode and dark mode changes.

We can work around this limitation by using a *theme*, which is a resource dictionary with a combined color palette and set of styles bundled up together. Because a theme doesn't depend on `DynamicResource`, the styles in the theme can use `AppThemeBinding`. Figure 10.11 shows an example of a theme.

Figure 10.11 A theme contains both colors and styles. If we want to change the theme, we can swap out the whole theme rather than just changing the color palette. Because it is using `StaticResource` rather than `DynamicResource`, it can also use `AppThemeBinding` and support light and dark mode.



If we want to change the theme, we can load a new one which will contain both colors and styles. Because we're not just swapping out the colors, we have no dependence on `DynamicResource`, so we can happily use `AppThemeBinding` to support light mode and dark mode.

For MauiStockTake, Mildred has asked that we provide a sandy theme to go alongside the default theme, and for both to support automatic light mode and dark mode in response to OS changes.

Start by creating two new resource dictionaries, one called `DefaultTheme` and

another called SandyTheme. You can do this in Visual Studio by selecting the .NET MAUI Resource Dictionary (XAML) template, or using the dotnet cli with the following command:

```
dotnet new maui-dict-xaml -na MauiStockTake.UI.Resources.Themes -
```

IMPORTANT

Make sure you use the template, rather than just creating a XAML file. When you use the template, it also includes a C# code-behind which, critically, calls `InitializeComponent()` in the constructor. This is required, as per the documentation linked above.

With these files created, we can build the themes by combining our existing color palettes and styles. **Copy everything from the Colors.xaml file**, between the `<ResourceDictionary...>...</ResourceDictionary>` tags (so all the color and brush definitions) to the top of the `DefaultTheme.xaml` file, between the `ResourceDictionary` tags in there. **Next, copy all the styles from the Styles.xaml file** and paste them into `DefaultTheme.xaml`, after the color definitions and before the closing `ResourceDictionary` tag.

Repeat this process to create the Sandy theme (copy the colors and styles into the `SandyTheme.xaml` resource dictionary). The two themes are currently identical, so let's update the Sandy theme so that it's got the right colors. We don't need to change much; update the `SandyTheme.xaml` file with the colors shown in listing 10.6.

Listing 10.6 The colors to update for the Sandy theme

```
<Color x:Key="Primary">#6b433b</Color>
<Color x:Key="PrimaryDark">#d9ceb6</Color>
<Color x:Key="Secondary">#ede8dd</Color>
<Color x:Key="PrimaryBackground">#cdb08a</Color>
<Color x:Key="SecondaryBackground">#dadccb</Color>
<Color x:Key="PrimaryDarkBackground">#251f13</Color>
<Color x:Key="SecondaryDarkBackground">#453b24</Color>
```

Now that we've got our two themes, we need to set one as default and provide the user with a way to switch between them. We can do the first one

by removing the `Colors.xaml` and `Styles.xaml` files from the resource dictionary in `App.xaml`. And, instead, replace them both with the new default theme:

```
<ResourceDictionary Source="Resources/Themes/DefaultTheme.xaml" /
```

Run the app for a quick check to make sure everything works the same as before. We haven't changed anything qualitative yet, we've just combined the styles and colors into a theme, so you shouldn't notice any difference.

We'll add a `MenuItem` to the `Shell` to allow the user to change the theme. **Download the `icon_palette.svg` file from this chapter's folder in the book's online resources and import it into the `Resources/Images` folder in `MauiStockTake.UI`; we will use this as the icon for this menu item.** We've already got a `MenuItem` for logging out, add the `MenuItem` in listing 10.7 to `AppShell.xaml` before the logout menu item:

Listing 10.7 The change theme menu item

```
<MenuItem IconImageSource="icon_palette.png"
          x:Name="ThemeMenuItem"
          Clicked="ThemeMenuItem_Clicked"/>
```

For this `MenuItem`, we've set the `IconImageSource` property, added an event handler, and given it a name. We haven't set the `Text` property as it will need to change depending on what the current theme is. We'll set its initial value in the `AppShell.xaml.cs` constructor and then change it as needed in the event handler. **Add this line to the `AppShell.xaml.cs` constructor:**

```
ThemeMenuItem.Text = "Switch to Sandy Theme";
```

To track which theme is currently in use, let's add an enum with all the available themes. We only have two for now, but this will let us easily add more in the future. If we add more, it's a good idea to provide a settings page where the user can choose; our current approach will only allow toggling between two themes, but that's fine for now.

In `MauiStockTake.UI`, add a file called `Theme.cs` in the `Helpers` folder, and add the code from listing 10.8.

Listing 10.8 The Theme enum

```
namespace MauiStockTake.UI.Helpers;  
public enum Theme  
{  
    Default,  
    Sandy  
}
```

To track the theme currently in use, add a static property to the App class. Add the following to App.xaml.cs.

```
public static Theme Theme { get; set; } = Theme.Default;
```

This gives us a static instance of the Theme enum with the default value already set. The final step to managing the theme is to write the event handler. In AppShell.xaml.cs, we'll add an event handler called ThemeMenuItem_Clicked (we've already specified this in the XAML) which will perform the following steps:

1. Check the static enum to see which theme is currently in use
2. Set the enum to the other value
3. Update the text of the menu item
4. Load the merged resource dictionary from the App class
5. Clear the dictionary
6. Load the alternate theme

The code for this event handler is shown in listing 10.9.

Listing 10.9 The ThemeMenuItem_Clicked event handler

```
private void ThemeMenuItem_Clicked(object sender, EventArgs e)  
{  
    if (App.Theme == Theme.Default)  
    {  
        App.Theme = Theme.Sandy;  
        ThemeMenuItem.Text = "Switch to Default Theme";  
        ICollection<ResourceDictionary> mergedDictionaries = App.  
        if (mergedDictionaries != null)  
        {  
            mergedDictionaries.Clear(); #B  
            mergedDictionaries.Add(new SandyTheme()); #C
```

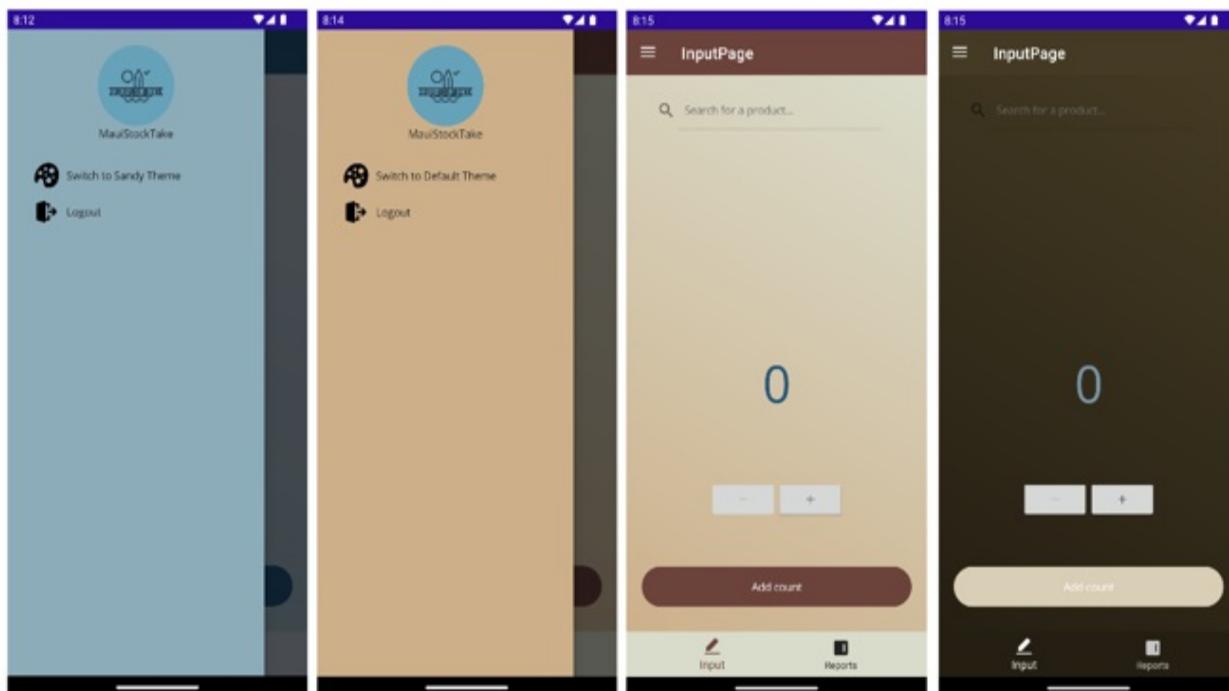
```

        }
    }
    else
    {
        App.Theme = Theme.Default;
        ThemeMenuItem.Text = "Switch to Sandy Theme";
        ICollection<ResourceDictionary> mergedDictionaries = Appl
        if (mergedDictionaries != null)
        {
            mergedDictionaries.Clear();
            mergedDictionaries.Add(new DefaultTheme());
        }
    }
}

```

Run the app now and it should start with the default theme. If you open the menu, you'll see the new switch theme menu item, and you should be able to change between the two different themes, and have them both respond to light and dark mode. Figure 10.12 shows these themes on Android.

Figure 10.12 The flyout now has a theme switching menu item. Note how the text changes depending on the currently selected theme. Note also how the Sandy theme responds to light and dark mode changes, just like the default theme does.



We've now got two themes, both using `AppThemeBinding` to support light and dark mode, that can be switched out at runtime. This is a somewhat heavy-handed approach, and if you don't need to support *both* light and dark mode *and* multiple color palettes, you're better off using *either* `AppThemeBinding` *or* `DynamicResource`. But this approach gives you both, which the other approaches do not.

10.2 Responding to State Changes

The UI we've built so far in this book has been relatively static; meaning once the UI has been defined, it doesn't change. But there are cases where we want the UI to be more dynamic, and change in response to changes in the state of the app.

This is what is meant by UI behavior. For example, in chapter 7 we looked at a `Switch` as an example, and how its background color could change in response to its state. In .NET MAUI, there are a few ways that we can make our UI dynamic in response to state changes.

Orientation Changes

The most efficient way to respond to orientation changes is to design your apps to be responsive. This means using layouts that can adapt to different orientations and screen sizes, which can be accomplished using the layouts provided in .NET MAUI.

However, if you want to explicitly change aspects of a page depending on orientation, the best approach is to use the `OnSizeAllocated` method we mentioned in chapter 7. You can override this method and compare the passed-in height and width values to determine the orientation.

What you do from there is up to you: you could hide or show different views, change font sizes, etc. Another approach is to set the value of an enum and bind properties in your XAML to it so that they can respond to orientation changes. The benefit of this approach is that it works for window resizing too. If we were to use this with the MauiCalc app that we built in chapter 5, we could show a scientific layout on mobile when it's in landscape, and on

desktop when the window is wider than it is taller (or wider than a minimum required width), and basic mode on mobile when the device is in portrait, or on desktop with a taller, rather than wider, window.

10.2.1 Triggers

In the `InputViewModel` in `MauiStockTake`, we've got a validation check in the `AddCount` method to ensure the user has selected a product before allowing them to submit the count. This is a good check that prevents saving bad data, but there are two things we can improve. The first is to add a check to prevent submitting a count of zero. The second is that, rather than waiting until the user tries to submit bad data and then showing a dialog, we can just disable the submit button until the data is good to go. And we can achieve both changes with *triggers*.

In .NET MAUI, triggers allow you to define, in XAML, a way for your UI to respond to events or state changes. Triggers use `Setter` to define the property and value that you want to control, just like in a `Style`. Triggers come in a few different flavors, but the two you are most likely to use are:

- **Property triggers:** Property triggers let you define a change in the appearance of a control when one of the properties on that control meets a certain condition. A property trigger would work well for the `Switch` example mentioned above.
- **Data triggers:** Data triggers let you define a change in the appearance of a control when a property on *any* control in the view, or property in the binding context, has a certain value. We'll use data triggers for the UX changes we're making to `MauiStockTake`.

The important thing to note about these triggers is that they will apply the specified modification to a control when the specified condition is met, and revert that modification when that condition is no longer satisfied.

NOTE

this is different to event triggers, which do not revert the change. You can read more about the full range of triggers and their differences at the

documentation here:

<https://learn.microsoft.com/dotnet/maui/fundamentals/triggers>

Let's start by removing the existing check. In `InputViewModel`, in the `AddCount` method, delete the `if` block that checks whether `SelectedProduct` is null. Now let's add our two data triggers to the `InputPage.xaml` file.

For both the `Stepper` and `Button` controls, we can expand them so they are no longer self-closing tags, and for each of them we can declare their `DataTriggers` collection. We'll add a data trigger with a `Setter` that sets the `IsEnabled` property to `False` to both.

With a trigger, just like with a `Style`, you need to define a target type, so these will be different for each one. We also need to provide a binding (the source data that the data trigger is bound to) and a value (the condition for the bound property to meet in order to activate the trigger). For the `Stepper`, this will be the `SelectedProduct` property of the `ViewModel`, and for the `Button` it will be the `Count` property.

The `Count` property is already bound to the UI (there is a `Label` that shows its value), so it already has an explicit backing field and its setter calls the `OnPropertyChanged` method. We need to do the same thing for the `SelectedProduct` property, so let's do this first. Listing 10.10. shows the updated `SelectedProduct` property declaration.

Listing 10.10 The updated SelectedProduct declaration

```
private ProductDto _selectedProduct;      #A
public ProductDto SelectedProduct          #B
{
    get => _selectedProduct;
    set
    {
        _selectedProduct = value;
        OnPropertyChanged();           #C
    }
}
```

Now that we've got a property that raises a property changed event, we can bind the UI to it. For the data trigger we're adding to the `Stepper`, we've set

the `IsEnabled` property to false. Let's bind that to the `SelectedProduct` property to activate when its value is null. That way, the Stepper will be disabled when no product is selected, and will be enabled once the user selects a product.

Because we're binding to a value that can be null (in fact it always will be when we set the binding), we can use a *binding fallback* to set the initial value when the bound property is null. Binding fallbacks are a useful tool that you can read more about in the .NET MAUI documentation here: <https://learn.microsoft.com/dotnet/maui/fundamentals/data-binding/binding-fallbacks>; but understanding them is not essential for now.

Listing 10.11 shows the updated code for the Stepper in `InputPage.xaml`.

Listing 10.11 The updated Stepper code in InputPage.xaml

```
<Stepper Grid.Row="3"
    HorizontalOptions="Center"
    VerticalOptions="Center"
    Value="{Binding Count}">
<Stepper.Triggers>          #A
    <DataTrigger TargetType="Stepper"           #B
        Binding="{Binding SelectedProduct, TargetNullValue=""}"
        Value="">                         #D
        <Setter Property="IsEnabled"             #E
            Value="False" />                  #F
    </DataTrigger>
</Stepper.Triggers>
</Stepper>
```

If you run the app now, you'll see that the Stepper on the `InputPage` is disabled until you search for a product and select one from the list.

Let's add the data trigger for the Button now. This will be almost identical to the data trigger we added to the Stepper; the differences will be the target type, the binding and the value. The target type will be `Button` instead of `Stepper`, and the binding will be the `Count` property. The `Count` property is of type `int` which has a default value of zero, so the value in the data trigger will be 0. Listing 10.12 shows the code for the updated Button in the `InputPage.xaml` file.

Listing 10.12 The updated Button code in InputPage.xaml

```
<Button Grid.Row="4"
        Text="Add count"
        Command="{Binding AddCountCommand}>
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Count}"          #A
                     Value="0">                      #B
            <Setter Property="IsEnabled"
                     Value="False"/>
        </DataTrigger>
    </Button.Triggers>
</Button>
```

With these changes made, you can run the app and should see that the Stepper is deactivated until you've selected a product. The Button to add the count is disabled until the Count is any number other than zero, and consequently also disabled until a product is selected (a product must be selected to change the value of Count and enable the Button).

Triggers can do more than just enable and disable controls. As they use Setters, just like a Style, they can modify any aspect of a control or view. Also, just like with a Style, a trigger can have multiple Setters, and can be used to specify a condition that modifies several aspects of a control.

10.2.2 Visual State Manager

Triggers are a great way to dynamically change the appearance of controls in response to events or data changes in your app. Using triggers, you can combine Setters to customize a control based on different conditions.

Sometimes you can logically group Setters in a trigger to define the appearance of a control under a specified set of conditions. But in this case, a more efficient approach is to define a *visual state* for the control.

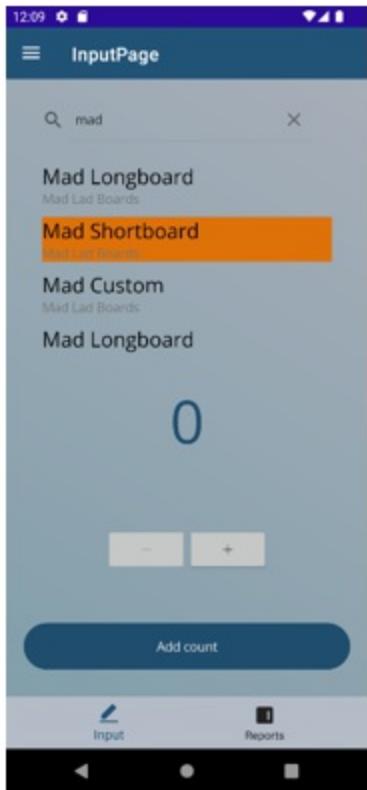
Visual states in .NET MAUI are managed by the *visual state manager* (VSM), a built-in tool that automatically alters the appearance of a view in response to the visual state that view is in. There are five built-in “common” visual states in .NET MAUI:

- Normal
- Disabled
- Focused
- Selected
- PointerOver

We've already seen these in action: the normal state is how controls look most of the time, disabled is what a control looks like when the `IsEnabled` property is set to false (as we saw in the previous section), and we've seen the selected state used in `CollectionView` a few times, notably in `MauiTodo` and in the product search results in `MauiStockTake`.

The selected state for the product search results `CollectionView` is a bit of a visual anomaly; it's not in keeping with Mildred's branding guidelines and looks out of place in all four of the app's themes. Figure 10.13 shows the current selected state in `MauiStockTake`.

Figure 10.13 The selected state in `CollectionView` makes the background of a selected item a shade of orange. The high contrast makes it easily distinguishable, but the color is not in keeping with the theme or brand of the app.



As versatile as triggers are, using them to modify the appearance of the selected item in a collection presents, at the very least, a logical challenge; but VSM makes this problem a lot simpler to solve. Let's use VSM to update the background color of the selected product in MauiStockTake.

VSM is applied in a Style, with VSM used as the Setter's property, rather than the property of a control. As it's embedded within a Style, it's applied using the same hierarchy that we looked at in section 10xx. We're going to modify the background color of `VerticalStackLayout` (the layout used in the `CollectionView` data template), so we want to make sure we scope it accordingly.

In `InputPage.xaml`, add a resource dictionary to the `CollectionView` used to display the product search results, and in here, add a style with a target type of `VerticalStackLayout`. In this Style, we're going to add a Setter, with the property set to `VisualStateManager.VisualStateGroups`.

Visual state groups are used to organise sets of visual states in VSM. As mentioned, .NET MAUI comes with a built-in visual state group called

`CommonStates`, that defines the five visual states listed above. Visual states in a visual state group are mutually exclusive, meaning only one visual state in a group can be active at a time. This means that a view cannot for example be both Normal and Selected.

Visual states in different groups can be active simultaneously, so if you need to, you can define multiple visual state groups. You can read more about defining custom visual states at the documentation here:

<https://learn.microsoft.com/dotnet/maui/user-interface/visual-states#define-custom-visual-states>.

Within the `Setter` in our `Style`, we first need to declare the visual state group list. We've only got the `CommonStates` visual state group as we haven't defined any custom states, so we'll embed this in the list. In this group, we'll need to define the Normal and Selected states. For Normal, we'll tell VSM to do nothing. This is needed because, unlike with a trigger, VSM does not revert the changes when the state is no longer active.

For the Selected state, we'll add a `Setter` for the `BackgroundColor` property, and set it to the `Secondary` static resource. Remember that this `Setter` is embedded within a `Style` that targets `VerticalStackLayout`, so the properties we have access to here are those of the specified target type. Listing 10.13 shows the updated code for the `CollectionView` in `InputPage.xaml` that includes the VSM `Style` (the non-relevant code has been omitted).

Listing 10.13 The CollectionView in InputPage.xaml with VSM

```
<CollectionView ...>
    <CollectionView.Resources> #A
        <Style TargetType="VerticalStackLayout"> #B
            <Setter Property="VisualStateManager.VisualStateGroup #C>
                <VisualStateGroupList> #D
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="Normal" />
                        <VisualState x:Name="Selected">
                            <VisualState.Setters>
                                <Setter Property="BackgroundColor
                                        Value="{StaticResource Secondary}" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
```

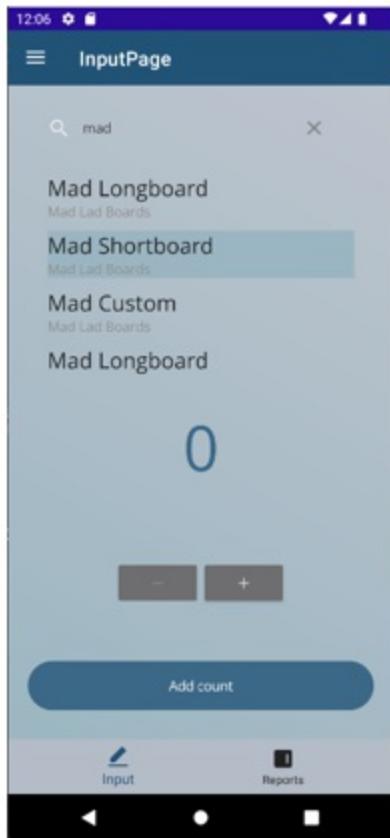
```

        </VisualStateGroupList>
    </Setter>
</Style>
</CollectionView.Resources>
<CollectionView.ItemsLayout>
    ...
</CollectionView.ItemsLayout>
<CollectionView.ItemTemplate>
    ...
</CollectionView.ItemTemplate>
</CollectionView>

```

Run the app now and search for a product. If you make a selection, you will see that the background color of the selected item is now updated. Figure 10.14 shows this for the default theme in light mode, but you should see the appropriate Secondary color for whichever theme you are running.

Figure 10.14 Visual State Manager has been used to update the Style for VerticalStackLayout, the layout used in the data template of the product search results CollectionView. For the Selected visual state, the BackgroundColor property is set to the Secondary static resource, providing a consistent look rather than the default.



Because VSM uses Styles, we can clean up the `InputPage` a bit by moving the Style out of the `CollectionView`'s resources and into the theme files. We don't want this to apply to every `VerticalStackLayout`, so we can use a key to make it an explicit Style. This has the added benefit that we can reuse it in other `CollectionViews` throughout our app.

Cut the `<Style...>...</Style>` tags and everything in between from the `CollectionView`'s resources and paste it into each of the theme files (wherever you like, but after the gradient brush we defined earlier is probably a good spot). Give it a key of `ProductSelector`.

Unlike with Triggers, changes that are applied when a visual state is entered are not reverted when the conditions for the visual state are no longer met. This means that we need to define the `Normal` visual state as well as any state that we want to customise. However, if we don't want to customise the `Normal` visual state (i.e., we want to retain the default appearance of the control), we can simply provide the `Normal` state without defining any setters.

Listing 10.14 shows how the Style should look in each of the theme files.

Listing 10.14 The `VerticalStackLayout` VSM Style in the theme files

```
<Style TargetType="VerticalStackLayout" x:Key="ProductSelector">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal" />
                <VisualState x:Name="Selected">
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                               Value="{StaticResource
                               </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

Now you can go back to the `CollectionView` in `InputPage.xaml` and remove the `<CollectionView.Resources>...</CollectionView.Resources>` tags, and everything in between. Add the explicit Style to the `VerticalStackLayout` in

the `CollectionView`'s data template. The `CollectionView` should now look like listing 10.15.

Listing 10.15 The final `CollectionView` in `InputPage.xaml`

```
<CollectionView ...> #A
    <CollectionView.ItemsLayout>
        ...
    </CollectionView.ItemsLayout>
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <VerticalStackLayout Style="{StaticResource ProductSe
                ...
            </VerticalStackLayout>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Run the app again and make sure you still get the Secondary background color on selected products in the `CollectionView`. By moving the `Style` and making it explicit, we've cleaned up the UI code and made it reusable, which helps to maintain the consistent look and feel throughout the app.

10.3 Multi-Platform Apps

.NET MAUI apps can run on a huge range of devices: laptop and desktop computers, phones, tablets, watches, and TVs. Sometimes the apps that you build will be targeted to a specific category of device, but often your app will need to work across many of these different device *idioms*.

Different device idioms have different UX paradigms. For example, touch and swipe gestures are ubiquitous on mobile devices, and while you can often achieve the same result with a mouse click and drag, these may not be as implicitly discoverable to desktop users, who expect to interact with your app in different ways.

There are different approaches to providing different UX for different device idioms. You can get the current idiom in code at any time by calling `DeviceInfo.Current.Idiom`. This returns a custom struct that is essentially an enum (and can be treated as such), that will give a value of `Phone`, `Tablet`,

Desktop, TV, Watch or Unknown. You can then use this to execute idiom-dependent logic or manipulate your UI as required. Brady Stroud has a cool sample showing this approach, using the idiom to load a different view for desktop vs all other platforms. You can see this in action in his GitHub repo here: <https://github.com/bradystroud/MauiMail>.

Another approach is to use compiler directives, which don't support different idioms but do let you specify a platform (note you can use `DeviceInfo.Current.Platform` to get the platform too). You can see this approach in use in David Ortinau's WeatherTwentyOne sample, available in his GitHub repo here: <https://github.com/davidortinau/WeatherTwentyOne>.

This can provide a more granular approach but can also prevent code that is only intended to run on one platform from being compiled to other platforms. This is useful for apps like Verinote, mentioned in chapter 1, that have functionality available on desktop that shouldn't be available on a mobile device; someone attempting to circumvent security measures can't access functionality that simply isn't there.

In this section, we're going to focus on the layout and how you can customize the appearance of the app for different platforms in XAML.

10.3.1 Adding the report page

Let's start by adding the report page. When we call the inventory from the API, there's a lot of information that might be useful to a desktop user but would overwhelm a small screen, and we want to optimize the screen real-estate we have for each platform. So the report page will need to behave differently on different device idioms.

Adding the ReportViewModel

We'll continue to follow the MVVM pattern with the `ReportPage`, so we'll put the functionality into a `ViewModel`. By Following the `ViewModel first` approach (see sidebar *View First vs ViewModel First* in chapter 9), we can build out this functionality (and even write tests for it) before even adding the UI.

In the ViewModels folder, add a class called `ReportViewModel` that inherits `BaseViewModel`. In this ViewModel, we'll need an `ObservableCollection` of type `InventoryItemDto` to bind the UI to, and we'll need to inject the `IInventoryService` into the constructor so that we can call the API to get the current inventory.

Add a method called `Init`; we'll call this from the `ReportPage`'s `OnAppearing` method, and add another method called `Refresh` that calls the `IInventoryService`, clears and then populates the `ObservableCollection`, and sets the `IsLoading` property from the `BaseViewModel` as required.

In the `Init` method, we can check a flag called `initialized` that has a default value of `false`. If it's `true`, we can just return from the method, otherwise we can call `Refresh`. This will let us re-use the `Refresh` method, which we'll need to do later in this section, and also if we later want to add pull-to-refresh.

This technique could be useful for other pages, so **add the initialised bool to the BaseViewModel and mark it as protected** (so it can be accessed from derived types). Listing 10.16 shows the code for the `ReportViewModel` class.

Listing 10.16 ReportViewModel.cs

```
using System.Collections.ObjectModel;
using MauiStockTake.Shared.Inventory.Queries;

namespace MauiStockTake.UI.ViewModels;
public class ReportViewModel : BaseViewModel
{
    private readonly IInventoryService _inventoryService;

    public ObservableCollection<InventoryItemDto> Inventory { get; }

    public ReportViewModel(IInventoryService inventoryService)
    {
        _inventoryService = inventoryService;
        IsLoading = true;
    }

    public async Task Init()
    {
        if (initialised)
```

```

        return;

    initialised = true;

    await Refresh();
}

private async Task Refresh()
{
    IsLoading = true;
    Inventory.Clear();

    var inventory = await _inventoryService.GetInventory();

    foreach (var item in inventory)
    {
        Inventory.Add(item);
    }

    IsLoading = false;
}
}

```

Now that we've added the `ReportViewModel`, let's register it in the service collection so that we can inject it into our report page. In `MauiProgram.cs`, after the line that registers the `InputViewModel`, add a line to register the `ReportViewModel`:

```
builder.Services.AddTransient<ReportViewModel>();
```

That's all the logic we need to add for now, so let's move on to the UI.

Adding the ReportPage UI

We've already got the `ReportPage` in `MauiStockTake`, but it's blank. Before we add the UI, let's add a field for the `ReportViewModel` to the code behind, and in the constructor inject the `ViewModel`, assign the page's `Navigation` property to the `ViewModel`'s `Navigation` property, and set the `ViewModel` as the binding context. Then, override the base `OnAppearing` method, make it `async`, and call the `ViewModel`'s `Init` method. Listing 10.17 shows the updated code for `ReportPage.xaml.cs`.

Listing 10.17 ReportPage.xaml.cs

```
namespace MauiStockTake.UI.Pages;

public partial class ReportPage : ContentPage
{
    private readonly ReportViewModel _viewModel;

    public ReportPage(ReportViewModel viewModel)
    {
        InitializeComponent();
        viewModel.Navigation = Navigation;
        _viewModel = viewModel;
        BindingContext = _viewModel;
    }

    protected override async void OnAppearing()
    {
        base.OnAppearing();

        await _viewModel.Init();
    }
}
```

We're almost ready to build the UI, but as `ReportPage` no longer has a default constructor, we need to register it for dependency injection. In `MauiProgram.cs`, after the line that registers the `InputPage`, add a line to register the `ReportPage`:

```
builder.Services.AddTransient<ReportPage>();
```

Now that we've wired everything up, let's add the UI. The main layout for this page will be a `CollectionView`, bound to the `Inventory` property of the `ViewModel`, but we'll wrap it in a `Grid` so that we can overlay an `ActivityIndicator` bound to the `IsLoading` (inherited) property on the `ViewModel`.

We'll explicitly set the `CollectionView`'s `ItemsLayout` property as `LinearItemLayout` so that we can specify the `ItemSpacing` as 30, to give us a bit of space between inventory items. Then, in the `DataTemplate`, we'll use a `Border` and `Shadow` to give us a card like effect, and then a `Grid` to layout the product name and the number currently counted in inventory.

We'll use AppThemeBinding for the various colors on the card so that they work with both themes in light and dark mode. Listing 10.18 shows the updated ReportPage.xaml file.

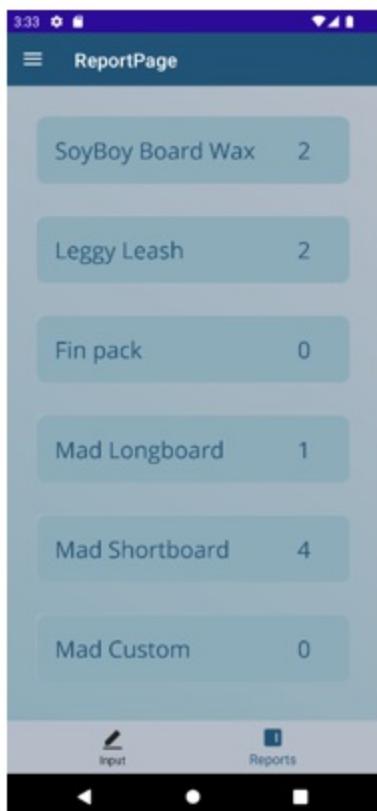
Listing 10.18 ReportPage.xaml

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiStockTake.UI.Pages.ReportPage"
    Title="ReportPage">
    <Grid>
        <ActivityIndicator HorizontalOptions="Center"
            VerticalOptions="Center"
            IsEnabled="True"
            IsRunning="True"
            IsVisible="{Binding IsLoading}"/>
        <CollectionView HorizontalOptions="Center"
            Margin="30"
            ItemsSource="{Binding Inventory}">
            <CollectionView.ItemsLayout>
                <LinearItemsLayout ItemSpacing="30"
                    Orientation="Vertical"/>
            </CollectionView.ItemsLayout>
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <Border StrokeShape="RoundRectangle 10"
                        Stroke="Transparent"
                        BackgroundColor="{AppThemeBinding Lig
                            <Grid ColumnDefinitions="4*, *"
                                Margin="20">
                                    <Label Grid.Column="0"
                                        TextColor="{AppThemeBinding Li
                                            FontSize="24"
                                            Text="{Binding ProductName}"/>
                                    <Label Grid.Column="1"
                                        FontSize="24"
                                        TextColor="{AppThemeBinding Li
                                            HorizontalTextAlignment="Cente
                                            Text="{Binding Count}"/>
                                </Grid>
                                <Border.Shadow>
                                    <Shadow Brush="{AppThemeBinding Light
                                        Offset="-5, -5"
                                        Radius="10"
                                        Opacity="0.8"/>
                            </Border.Shadow>
                        </Border>
                    </DataTemplate>
                </CollectionView.ItemTemplate>
            </CollectionView>
        </Grid>
    </ContentPage>
```

```
        </Border .Shadow>
    </Border>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
</Grid>
</ContentPage>
```

We're not doing anything new here; we've seen similar card views in other apps we've built, and we've seen AppThemeBinding earlier in this chapter. This gives us a basic layout that will work well on mobile. If you run the app now and open the report page, you should see something similar to figure 10.15.

Figure 10.15 The ReportPage running on Android. The card layout works on a small screen and presents only the minimal information required, which is product name and current inventory count.



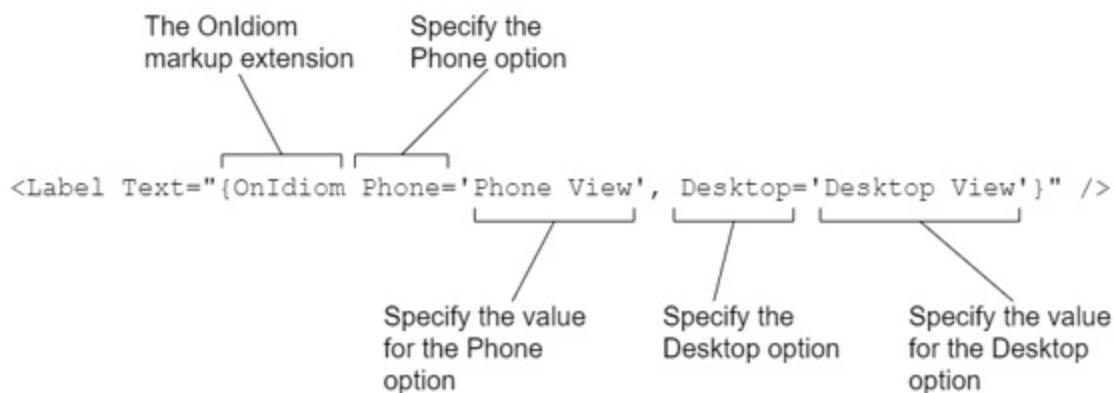
This layout works well on a phone, but it has two problems on desktop (or tablet too). The first is that it looks terrible (you can run it now on Windows

or macOS to see how it looks on desktop). The second is that the minimal information that we've pared down to for the mobile display isn't an efficient use of screen real-estate.

10.3.2 Multi-platform layouts

We've seen how you can use the `DeviceInfo` API to get information about the current platform or idiom your app is running on (in fact you can get a lot more information from the API too). This is a good approach if you need this in your C# code, and in XAML we can use markup extensions, such as `OnIdiom`. Using `OnIdiom`, you can specify different values, directly in XAML for any property of any view, for different platforms. Let's look at the following example:

Figure 10.16 The `OnIdiom` markup extension is used to specify different values depending on the device idiom. In this example, different strings are assigned to the `Text` property of a label depending on whether it is running on a desktop or a phone.



Like the `DeviceIdiom` struct, `OnIdiom` lets you specify values for Phone, Tablet, Desktop, Watch or TV, but instead of unknown, it has an option for Default. So you don't necessarily need to specify a value of each idiom; you can specify your standard value and just change it for the idiom you want to modify.

Let's start by updating some of the spacing. In the `LinearItemsLayout` tag, change it from 30 to 10 for desktop, 30 for phone, and a default value of 20. That line should now look like this:

```
<LinearItemsLayout ItemSpacing="{OnIdiom Desktop=10, Phone=30, De
```

For the `Margin` property in the `Grid`, change from 20 to a default value of 0 and 20 on phone. That line should now look like this:

```
Margin="{OnIdiom Phone=20, Default=0}">
```

Next, let's change the `BackgroundColor` of the `Border` so that it only shows the bound color on phone. Wrap current `AppThemeBinding` as a value for `Phone` and set the default to transparent. That line should now look like this:

```
BackgroundColor="{OnIdiom Phone={AppThemeBinding Light={StaticRes
```

This will give us a better layout for desktop, but ideally, we'd display the information in tabular form, and expose more of the data. We can use the `OnIdiom` markup extension to specify a different number of columns: two for phone, as we have been doing, but five for desktop so that we can show the product name, manufacturer name, count, date counted and who counted it.

On phone, as we only have two columns, we want the second column to show the count, but on desktop, as we have more columns to play with, we can move the count further along and show the manufacturer name in the second column, by specifying a different binding depending on the idiom.

To add the extra columns, we can just add the `Labels` we need, placing them in the appropriate columns, and we can use `OnIdiom` to set the `IsVisible` property depending on whether we're on phone or desktop.

As we're now displaying this information as a table, we should add some headers. We can specify the `CollectionView.Header` property, and in here we'll add a `Grid` with five columns, with the same definitions as the desktop view in the data template. We'll use `OnIdiom` to make the whole `Grid` invisible on phone but visible on desktop.

Inside this `Grid`, we can add five `Labels`, which will be almost identical to the `Labels` in the data template. The differences will be that the `FontAttributes` will be set to bold, and the column headers will be hard-coded.

The last bit of flair to add for desktop is to add a data trigger to the `Label` that shows the date counted. As the value is a `DateTime`, the DTO has a default value of `DateTime.Minimum` if no explicit value is returned from the database. This will occur if there have been no counts recorded for that item, so we can use a data trigger to set the `Text` property of the `Label` to “No stock counted” when this occurs.

NOTE

you could achieve the same result by binding the data trigger to the `Count` property and using 0 as the value. But this approach demonstrates the versatility of data triggers.

Listing 10.19 shows the updated code for `ReportPage.xaml`, including all the `OnIdiom` markup extensions and the added data trigger.

Listing 10.19 The final `ReportPage.xaml` code

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage ...>
    <Grid>
        <ActivityIndicator .../>
        <CollectionView ...>
            <CollectionView.ItemsLayout>
                <LinearItemsLayout ItemSpacing="{OnIdiom Desktop=10,
                                                Orientation=Vertical"/>
            </CollectionView.ItemsLayout>
            <CollectionView.Header>
                <Grid IsVisible="{OnIdiom Phone=False, Desktop=True}"
                      ColumnDefinitions="2*, 2*, *, 2*, 2*"
                      >
                    <Label Grid.Column="0"
                          Text="Product"
                          FontSize="24"
                          FontAttributes="Bold"/>
                    <Label Grid.Column="1"
                          Text="Manufacturer"
                          FontSize="24"
                          FontAttributes="Bold"/>
                    <Label Grid.Column="2"
                          Text="Count"
                          FontSize="24"
                          FontAttributes="Bold"/>
                    <Label Grid.Column="3"
                          Text="Last Counted"
                          FontSize="24"
                          FontAttributes="Bold"/>
                </Grid>
            </CollectionView.Header>
            <CollectionView.Content>
                <StackLayout Orientation="Vertical">
                    <Image Source="~/Assets/icon-product.png" />
                    <Text Text="Product" />
                    <Text Text="Manufacturer" />
                    <Text Text="Count" />
                    <Text Text="Last Counted" />
                </StackLayout>
            </CollectionView.Content>
        </CollectionView>
    </Grid>
</ContentPage>
```

```

        Text="Counted By"
        FontSize="24"
        FontAttributes="Bold"/>
    <Label Grid.Column="4"
        Text="Counted On"
        FontSize="24"
        FontAttributes="Bold"/>

```

</Grid>

```

</CollectionView.Header>
<CollectionView.ItemTemplate>
    <DataTemplate>
        <Border StrokeShape="RoundRectangle 10"
            Stroke="Transparent"
            BackgroundColor="{OnIdiom Phone={AppT
                PrimaryDarkBackground}}, Default=Tran
        <Grid ColumnDefinitions="{OnIdiom Phone='
                        Margin="{OnIdiom Phone=20, Default=
        <Label Grid.Column="0"
            TextColor="{AppThemeBinding Li
            FontSize="24"
            Text="{Binding ProductName}"/>
        <Label Grid.Column="1"
            FontSize="24"
            TextColor="{AppThemeBinding Li
            HorizontalTextAlignment="{OnId
            Text="{OnIdiom Phone={Binding
        <Label Grid.Column="2"
            TextColor="{AppThemeBinding Li
            FontSize="24"
            Text="{Binding Count}"
            IsVisible="{OnIdiom Phone=Fals
        <Label Grid.Column="3"
            FontSize="24"
            TextColor="{AppThemeBinding Li
            Text="{Binding CountedByName}"
            IsVisible="{OnIdiom Phone=Fals
        <Label Grid.Column="4"
            FontSize="24"
            TextColor="{AppThemeBinding Li
            Text="{Binding CountedAt}"
            IsVisible="{OnIdiom Phone=Fals

```

<Label.Triggers>

```

        <DataTrigger TargetType="Labe
            Binding="{Bindin
            Value="1/1/0001"
        <Setter Property="Text"
            Value="No stock c

```

```

        </DataTrigger>
    </Label.Triggers>
</Label>
</Grid>
<Border.Shadow>
    <Shadow Brush="{AppThemeBinding Light
        Offset="-5, -5"
        Radius="10"
        Opacity="0.8"/>
</Border.Shadow>
</Border>
</DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
</Grid>
</ContentPage>

```

Run the app now on desktop and go to the report page. You should see all the information displayed in a nice tabular view, like figure 10.17.

Figure 10.17 The ReportPage running on Windows. The OnIdiom markup extension is used to change the way the information is displayed, providing additional information and a tabular layout, compared to the simple card view shown on phones.



Product	Manufacturer	Count	Counted By	Counted On
SoyBoy Board Wax	Bobbie's Surf Supplies	2	administrator@localhost	11/3/2022 12:00:00 AM
Leggy Leash	Bobbie's Surf Supplies	2	administrator@localhost	1/1/0001 12:00:00 AM
Fin pack	Bobbie's Surf Supplies	0	No stock counted	No stock counted
Mad Longboard	Mad Lad Boards	1	administrator@localhost	1/1/0001 12:00:00 AM
Mad Shortboard	Mad Lad Boards	4	administrator@localhost	1/1/0001 12:00:00 AM
Mad Custom	Mad Lad Boards	0	No stock counted	No stock counted
Men's rashie	Natura Surf Fashion	0	No stock counted	No stock counted
Women's rashie	Natura Surf Fashion	0	No stock counted	No stock counted
Unisex signlet	Natura Surf Fashion	0	No stock counted	No stock counted

Run the app again on Android or iOS; you should note no change from before and it should look the same as in figure 10.yy.

If you look at the examples linked to at the start of this section, you can see how you can load different views for different idioms and platforms. Using the approach here, you can see that we can customise the display for different platforms and idioms, all within the same view.

10.3.3 Features of desktop apps

Providing a desktop-specific layout is the most important thing you can do to provide a good UX for desktop users, and this will give your app a professional feel that distinguishes it from “lift and shifted” mobile apps that have been ported to desktop.

But you can take this a step further by utilizing the UX paradigms users have come to expect on desktop OSes: fundamentally, menus and windows. Let’s finish off this chapter by seeing how you can add these to your .NET MAUI app.

Menus

Menus can be added to pages in an app by specifying the page’s `MenuBarItems` collection. Because the menu bar is only shown on desktop, you don’t need to use `OnIdiom` (or any other technique) to hide it on mobile or other idioms.

Menus are added at the top level by specifying a `MenuItem`, and these menus can be made up of `MenuFlyoutItems`, which display text and offer an action to execute, or `MenuFlyoutSubItems`, which let you group `MenuFlyoutItems` into sub-menus.

Let’s add a menu now to the report page. In the `ReportPage.xaml` file, add the `ContentPage.MenuBarItems` tags before the `Grid`, and add a `MenuItem` with `Text` property of `Help`. Add a single `MenuFlyoutItem` in here with a `Text` property of `About`.

`MenuFlyoutItem` has a `Clicked` event and a `Command` property. You can use either of these to execute an action when your user clicks on the menu item, with either an event handler in your code behind or a command in your

binding context.

Listing 10.20 shows the menu added to the report page.

Listing 10.20 The ReportPage.xaml menu

```
<ContentPage.MenuBarItems> #A
    <MenuBarItem Text="Help"> #B
        <MenuFlyoutItem Text="About" #C
            Command="ShowAboutPageCommand"/>
    </MenuBarItem>
</ContentPage.MenuBarItems>
```

You can run the app now if you like, and if you navigate to the report page, you'll see the Help menu displayed in the top left as in figure 10.18, next to the title bar. If you try to click on the About menu item now it won't work, but in the next section, we'll add the About page and get the app to display it in a window.

Figure 10.18 The ReportPage, but with a Help menu added. The Help menu has a single menu item called About.



Product	Manufacturer	Count	Counted By	Counted On
SoyBoy Board Wax	Bobbie's Surf Supplies	2	administrator@localhost	11/3/2022 12:00:00 AM
Leggy Leash	Bobbie's Surf Supplies	2	administrator@localhost	11/6/2022 12:00:00 AM
Fin pack	Bobbie's Surf Supplies	0	No stock counted	No stock counted
Mad Longboard	Mad Lad Boards	1	administrator@localhost	11/6/2022 12:00:00 AM
Mad Shortboard	Mad Lad Boards	4	administrator@localhost	11/6/2022 12:00:00 AM
Mad Custom	Mad Lad Boards	0	No stock counted	No stock counted
Men's rashie	Natura Surf Fashion	0	No stock counted	No stock counted
Women's rashie	Natura Surf Fashion	0	No stock counted	No stock counted
Unicoy sinner	Natura Surf Fashion	0	No stock counted	No stock counted

Windows

When you launch a .NET MAUI app on Windows or macOS, an initial window is created, and the app's `MainPage` is loaded into this window. In the `blankmaui` template, we assigned a content page to this property, and in the default template `Shell` is assigned to it, and this becomes the content of the app's main window.

But you can also create additional windows and assign your own pages to them programmatically. Let's add a page to make the About menu work. In the `Pages` folder in `MauiStockTake`, add a new XAML page called `AboutPage`. The content of this page should be a `VerticalStackLayout` with `spacing = 30` containing three `Labels`. All three `Labels` should be centered and have horizontal text alignment. Assign the first `Label`'s text property "Welcome to MauiStockTake", "v1.0" to the second, and "Copyright Mildred's Surf Shack 2022" to the third. Assign them the font sizes of `Header`, `Title` and `Large` respectively.

Now that we've got a page to show in the About window, let's add a method to the `ReportViewModel` to show it. We'll need to create a new instance of the `Window` type, and we can pass a `ContentPage` to its constructor (or directly assign one to its `Page` property). It's a good idea to set the `Title` property, and the `Height` and `Width` as well. When we're ready, we can call `Application.Current.OpenWindow` and pass in our `Window` instance.

The method to add to `ReportViewModel` should look like listing 10.21.

Listing 10.21 The `ShowAboutPage` method in `ReportViewModel`

```
public void ShowAboutPage()
{
    var newWindow = new Window(new AboutPage())
    {
        Title = "About",
        Width = 300,
        Height = 300
    };
    Application.Current.OpenWindow(newWindow);
}
```

Add the `ShowAboutPageCommand` property (of type `ICommand`) to the `ReportViewModel`:

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

Then, in the constructor assign the method to it via a new Command instance:

```
ShowAboutPageCommand = new Command(ShowAboutPage);
```

Windows are a staple of desktop apps, but they are also supported on iOS and Android too (depending on your system). What we've done so far will work fine for Windows and Android, but there are a couple more steps necessary to enable multi-windowing on macOS and iOS. The first thing we need to do is create a SceneDelegate. A Scene in iOS and Mac Catalyst is a running instance of your app and is responsible for managing your app's windows and UI.

Listing 10.22x shows the code for the SceneDelegate to add into the iOS and MacCatalyst platform folders in MauiStockTake. Take care to update the namespace accordingly.

Listing 10.22 The SceneDelegate file

```
using Foundation;

namespace MauiStockTake.UIPlatforms.[iOS/MacCatalyst];      #A

[Register("SceneDelegate")]
public class SceneDelegate : MauiUISceneDelegate
{
}
```

The final step is to update the `info.plist` file to declare that you're using multiple scenes. Add the key and dictionary in listing 10.23 to the `info.plist` file in both the iOS and MacCatalyst platform folders.

Listing 10.23 The Multi-window key to add to info.plist

```
<key>UIApplicationSceneManifest</key>
<dict>
    <key>UIApplicationSupportsMultipleScenes</key>
    <true/>
    <key>UISceneConfigurations</key>
    <dict>
        <key>UIWindowSceneSessionRoleApplication</key>
```

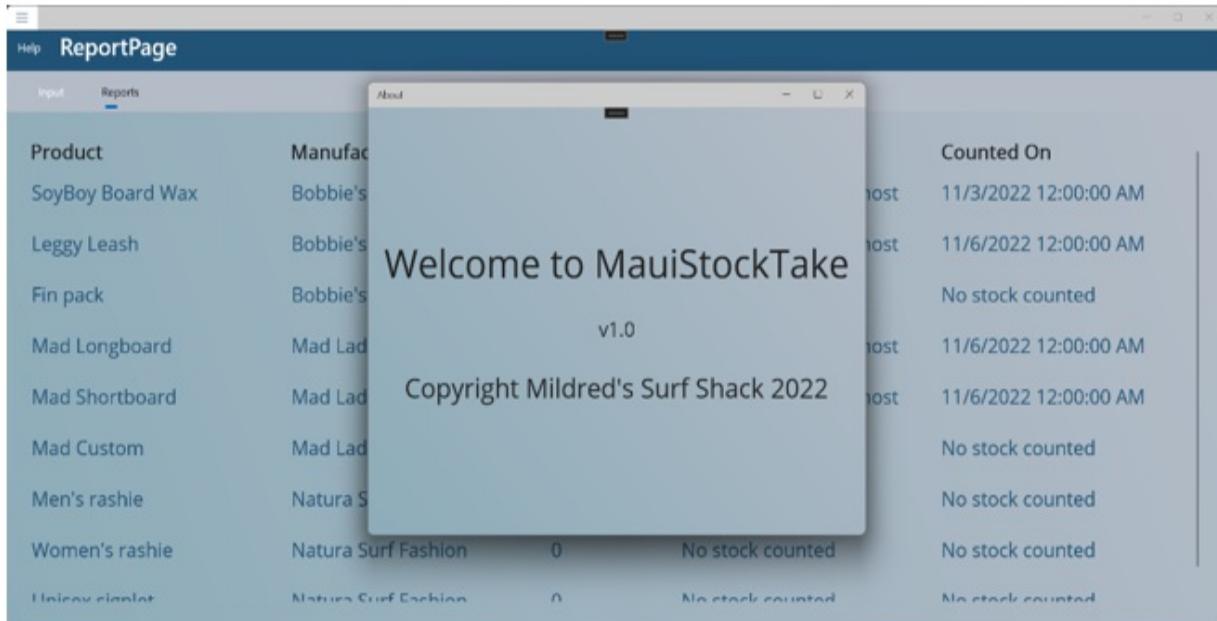
```

<array>
  <dict>
    <key>UISceneConfigurationName</key>
    <string>__MAUI_DEFAULT_SCENE_CONFIGURATION__</string>
    <key>UISceneDelegateClassName</key>
    <string>SceneDelegate</string>
  </dict>
</array>
</dict>
</dict>

```

Run the app now and go to the report page. Click on the Help menu, and then the About menu item. It should open your AboutPage in a new window, just like in figure 10.19:

Figure 10.19 The AboutPage for MauiStockTake shown in a new Window. It is just a ContentPage, like the other ContentPages in our app, but it's been assigned to the Page property of an instance of the Window class. This instance can then be shown using the OpenWindow method.



10.4 Summary

- Styles are collections of Setters that modify the values of properties on a control.
- Styles can be implicit or explicit. With implicit Styles, you can apply

visual changes to all instances of a control in your app. With explicit Styles you can limit the changes only to specific instances.

- A hierarchy is used to determine the final appearance of a control. App-wide Styles are applied first, but page-specific Styles take precedence. Layout or control specific Styles take higher precedence than page Styles, and explicit Styles override any other Style. Property values set directly on a control override any value in a Style.
- You can bind to the device's light or dark mode setting using `AppThemeBinding`. This makes it easy to provide light and dark mode views or your app.
- You can use `DynamicResource` to change themes and colors at runtime.
- Data triggers let you respond to changes in your app's state at runtime. You can change any property of any layout or control in response to a state or value change anywhere in your app.
- `VisualStateManager` lets you group Setters into a visual state. Visual states are collected in visual state groups. Five states called Common States come built in to .NET MAUI (Normal, Selected, Disabled, Focused and PointerOver), but you can define your own too.
- .NET MAUI apps can run on a range of different devices, including laptop and desktop computers, phones, TVs and watches. Which one of these the app is running on is called the device idiom.
- There are several ways to modify your app for different platforms. You can use compiler directives to control what code gets compiled to which platforms, and the `DeviceInfo` API can provide the current platform or idiom at runtime.
- The `OnIdiom` markup extension can let you define different views and layouts for different platforms, all within the same XAML file.
- Different UX paradigms are common on different idioms. With .NET MAUI, you can have menus and multi-windows in your apps to cater to desktop users.

11 Beyond the Basics: Custom Controls

This chapter covers

- Building reusable components with templated controls
- Creating your own bindable properties
- Modifying the default controls with handlers
- A recap on code sharing

.NET MAUI comes with enough controls to let you build almost any UI. Functionally, there is very little that you can't do using just the standard controls, and they are also highly customizable through styles and the various styling properties. But, sometimes, you need to go a little further.

In .NET MAUI you have a few ways to build or customize controls, from bundling controls together into a reusable component, to customizing the platform implementations that come in the box, to drawing your own controls and graphics with the `Microsoft.Maui.Graphics` library.

NOTE

`Microsoft.Maui.Graphics` is a powerful library capable of sophisticated image generation and manipulation. Drawing your own controls is only a small subset of what it's capable of. If you're interested in learning more, see: <https://learn.microsoft.com/dotnet/maui/user-interface/graphics/>

In this chapter, we'll look at the first two of these three approaches. We'll start by building our own control by re-using the built-in controls, and we'll also see how we can modify the way that .NET MAUI displays the built-in controls by default.

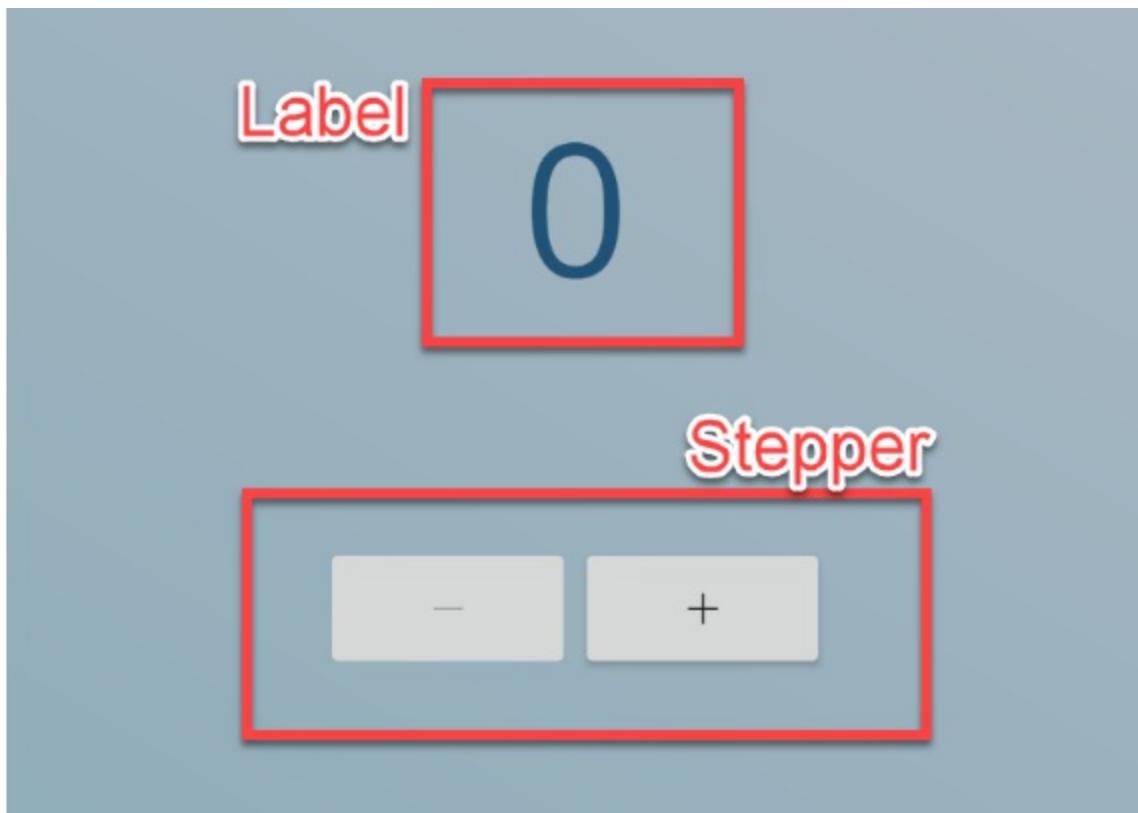
11.1 Using ContentView

Componentization is a core feature of every modern UI framework. While you can build your whole UI using the elementary controls that come in the box, a more efficient approach is to combine these controls into reusable components.

If you've worked with any modern UI framework before, you'll be familiar with this approach; it's used in Blazor, Angular, React, Flutter, and countless others. The terminology differs across frameworks, but it's usually some variation of 'reusable components'. In .NET MAUI, these reusable components are built using `ContentView`.

Let's look at an example from MauiStockTake. On the input page, we need to allow users to record how many of a specific item they have counted, and we need to display this count to the user. This functionality is provided by combining two controls, as shown in figure 11.1.

Figure 11.1 A Label and a Stepper are used to display the current count and allow the user to input the current count respectively. Anywhere else this functionality is required we need to add both controls again.



We have a UX problem with the app, as we mentioned earlier, which is that it's problematic for larger numbers, especially anything greater than 10. Can you imagine trying to enter 546 using just a stepper? We can build a custom control that will solve this UX problem.

A common solution to this problem is to provide an input field with a Stepper built into it, as in figure 11.2.

Figure 11.2 An input field for entering numerical values. The user can type a number in and use the up and down arrows to increase and decrease the value respectively.



This works well in forms on the web, and can work on desktop too, but it isn't particularly touch friendly, and therefore isn't a great choice for mobile. Instead, we can combine this with the existing design to create a custom stepper control with an editable value field built in, as shown in figure 11.3.

Figure 11.3 A templated control made up of two Buttons and an Entry. The Buttons can be used to increase or decrease the value of a bound property, and the Entry can be used to edit it directly.



We could just build these controls directly into the UI, and while dropping an Entry and a couple of Buttons onto a page as and when we need them isn't particularly laborious, building a templated control lets us reuse the UI and saves us having to solve the problem every time we encounter it. And the more complex the UI, the more value there is in making it a reusable component.

TIP

Be on the lookout for opportunities to bundle up parts of your UI for reuse.

Don't repeat yourself!

11.1.1 Building the Custom Stepper layout

Let's get started building the custom stepper control. **Add a folder to MauiStockTake.UI called Controls.** In here, we're going to add a .NET MAUI ContentView (XAML). You can do this from the context menu in Visual Studio or using the .NET CLI.

Add a new ContentView called MildredStepper. The layout for this control will be a Grid, with one row and three columns (one for each Button and one for the Entry in the middle). The width of the Button columns will be 50, and the width of the Entry column will be 120 (to accommodate larger numbers). We'll vertically and horizontally center everything and use a large font size (42 point) for the number, same as what we have now for the Label that displays the count.

We'll need event handlers on all three controls so that we can set a value property. The last thing we need to do is set MinimumWidthRequest for the Buttons to 50. This is set in the theme as 100 (from the default Style), so we need to override it so that the buttons can fit inside their columns. Listing 11.1 shows the code for the custom stepper's layout.

Listing 11.1 The custom stepper layout

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentView xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiStockTake.UI.Controls.MildredStepper">
    <Grid ColumnDefinitions="50,120,50">
        <Button Grid.Column="0"
            Text="-"
            Clicked="MinusButton_Clicked"
            x:Name="MinusButton"
            VerticalOptions="Center"
            HorizontalOptions="Center"
            MinimumWidthRequest="50"/>
        <Entry Grid.Column="1"
            x:Name="ValueEntry"
            FontSize="42"
            HorizontalTextAlignment="Center"
```

```

        TextChanged="ValueEntry_TextChanged"
        VerticalOptions="Center"
        HorizontalOptions="Center"/>
    <Button Grid.Column="2"
        Text="+"
        Clicked="PlusButton_Clicked"
        x:Name="PlusButton"
        VerticalOptions="Center"
        HorizontalOptions="Center"
        MinimumWidthRequest="50"/>
</Grid>
</ContentView>
```

The Buttons and the Entry all have event handlers defined, so we need to add these in the code behind. We'll need a property to store the value, and when the `TextChanged` event is fired, we'll need to get the `Text` value of the `Entry`, parse it to an `int` and assign it to the property. When one of the Buttons is clicked, we'll need to increase or decrease the value of the property, and then cast it to a `string` and assign it to the `Text` property of the `Entry`.

Listing 11.2 shows the code for `MildredStepper.xaml.cs` file.

Listing 11.2 The MildredStepper.xaml.cs file

```

namespace MauiStockTake.UI.Controls;

public partial class MildredStepper : ContentView
{
    public int Value { get; set; }

    public MildredStepper()
    {
        InitializeComponent();
        ValueEntry.Text = "0";
    }

    private void MinusButton_Clicked(object sender, EventArgs e)
    {
        Value--;
        ValueEntry.Text = Value.ToString();
    }

    private void PlusButton_Clicked(object sender, EventArgs e)
```

```

    {
        Value++;
        ValueEntry.Text = Value.ToString();
    }

    private void ValueEntry_TextChanged(object sender, TextChange
    {
        if (int.TryParse(e.NewTextValue, out var value))
        {
            Value = value;
        }
    }
}

```

Now that we've built our custom stepper control, we can use it in the input page. When adding things to our XAML files that are not part of the standard .NET MAUI controls and markup extensions, we need to add an XML namespace (just as we have done for Behaviors in the .NET MAUI Community Toolkit).

We've currently got two rows of the Grid allocated for this functionality (one for the Label and one for the Stepper). We only need one row with our custom control, so we'll need to adjust the row definitions, and the row allocation for the add count Button (it was previously in row 4, it will now be row 3).

Listing 11.3 shows the updates to make to the `InputPage.xaml` file.

Listing 11.3 The updated InputPage.xaml file

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xam
    x:Class="MauiStockTake.UI.Pages.InputPage"
    xmlns:controls="clr-namespace:MauiStockTake.UI.Contr
    Title="InputPage">
    <Grid Padding="20"
        RowDefinitions="*, 3*, 4*, 1*"> #B
        <SearchBar .../>

        <CollectionView Grid.Row="1" ...>
            ...
        </CollectionView>

```

```

<ActivityIndicator Grid.Row="1" .../>

<controls:MildredStepper Grid.Row="2"
    HorizontalOptions="Center"
    VerticalOptions="Center"/>

<Button Grid.Row="3" #D
    ...
</Button>
</Grid>
</ContentPage>

```

You can run the app now, and you'll see that the Label and Stepper have been replaced with our new custom control. You can click the plus and minus Buttons to increase and decrease the count, and you can click directly into the Entry to edit the number directly.

This is good so far, but the Stepper and Label that this control has replaced had bindings to the ViewModel; our custom control doesn't have that. This is a pretty important feature, and we'll fix that up in 11.2. But before we do, let's add a couple of UX improvements.

11.1.2 Improving the custom stepper's UX

There are a couple of small changes we can make to the custom stepper control that will significantly improve the UX. The first is that it's possible to enter a negative number, either via the Entry or using the Buttons. Let's start by putting a check on the minus Button to ensure it doesn't decrease the number to less than 0.

In the `MinusButton_Clicked` event handler, add a check to ensure the value is greater than 0 before decreasing it:

```

private void MinusButton_Clicked(object sender, EventArgs e)
{
    if (Value > 0)
    {
        Value--;
        ValueEntry.Text = Value.ToString();
    }
}

```

Once the number reaches 0, clicking on the minus button won't do anything. Let's add a simple check to the `Entry` event handler too. Before we parse the number, let's check to see if the new value begins with "-". If it does, we can reset the `Text` property of the `Entry` back to the current value and exit out of the event handler. Add the following code inside the event handler, before parsing the new value:

```
if (e.NewTextValue.StartsWith("-"))
{
    ValueEntry.Text = Value.ToString();
    return;
}
```

This will prevent users from accidentally trying to record a negative stock count.

Validation

Usually in a UI app you would validate all user input. What we've done here is *almost validation, but it's not quite, for one reason: we don't provide any feedback to the user.*

Validation can take many forms, for example ensuring an entered value is a number between a certain range, checking that it's in an approved list, verifying that it's an email address, and so on. With validation, a user is not prevented from entering an invalid number, but rather is informed that the entry is invalid and prevented from submitting the form.

We've done the last part already (disabled the add count button) but we're not giving the user any feedback. In .NET MAUI, you have a lot of options for implementing validation. We could expand what we've done already by adding a `Label` to the UI that displays a warning message, and only make it visible when the validation has failed.

Additional validation can be done with bindable properties, which we're going to look at in the next section. But the best way to get started is to look in the .NET MAUI Community Toolkit. The toolkit has a handful of common validations included as behaviors that are simple to implement in your .NET MAUI apps. To find out more, see:

<https://learn.microsoft.com/dotnet/communitytoolkit/maui/behaviors/>

The second UX improvement is to provide the user with the right keyboard when they enter values directly. When you run the app, if you go to edit the value in the custom stepper, you'll use the regular keyboard, which has a full set of keys and a small row of buttons along the top. We only want numbers entered here, and we can make this easier for the user by giving them a numeric keyboard. To do this, **set the Keyboard property of the Entry in the MildredStepper.xaml file to Numeric.** If you run the app again now, you will see a numeric keypad when editing this field.

Improving input UX in .NET MAUI

You can improve the UX in many cases by providing the right keyboard for your user. If you need the user to input numbers, use the numeric keyboard. The email keyboard provides quick access to the @ symbol and common top-level domains. The keyboard types you can use are Default, Chat, Email, Numeric, Plain, Telephone, Text and URL.

You can also improve input UX in other ways. For example, for a password field, you can set IsPassword to true, and this will mask the input as the user types it, transforming their entered characters into asterisks.

The .NET MAUI Community Toolkit includes a MaskedBehavior, which can be attached to an Entry to make the input match a specific pattern. This is particularly useful for credit card numbers or specific telephone number formats. You can find out more about it here:

<https://learn.microsoft.com/dotnet/communitytoolkit/maui/behaviors/masked-behavior>.

Note that these are UX improvements only; a user can still bypass them, which is why it's important to combine them with validation techniques mentioned above.

Our custom stepper control is now almost complete. The last thing we need to do is provide a way to get the value out of the control and into the parent page or layout.

11.2 Bindable Properties

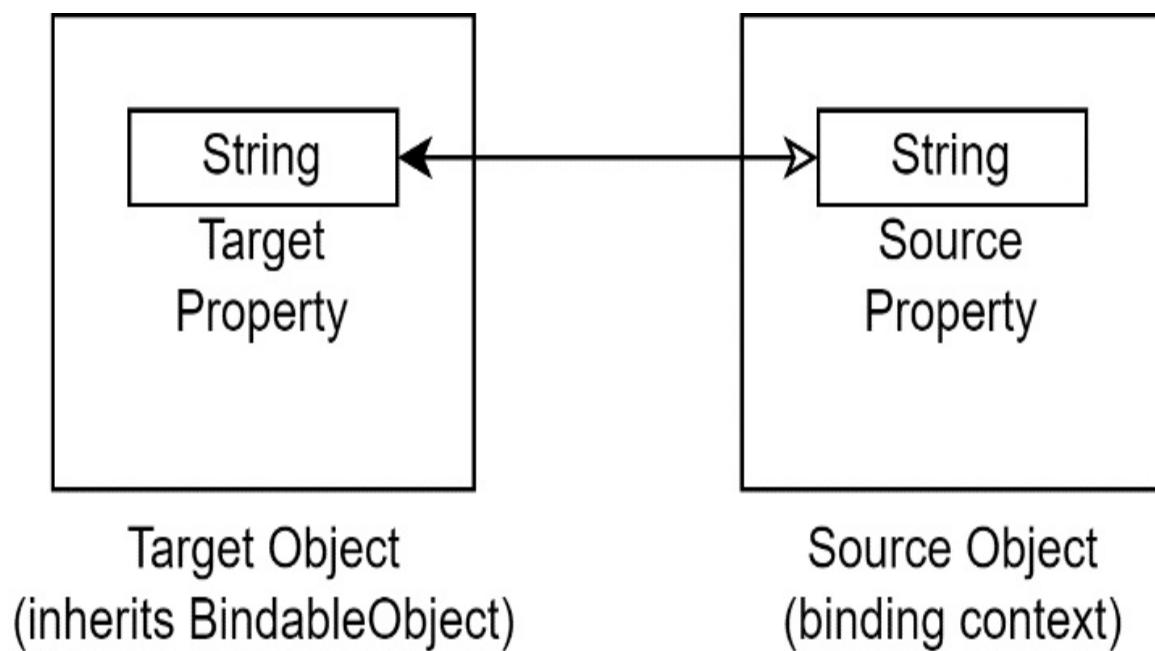
A core concept in .NET MAUI is *bindable properties*. These are an extension of the class properties you are familiar with in .NET (i.e., a public class member with a getter and setter) but with additional functionality that enables, among other things, data binding.

NOTE

Bindable properties provide a lot more functionality, and to get an idea of everything they can do, I recommend looking through the documentation here: <https://learn.microsoft.com/dotnet/maui/fundamentals/bindable-properties>.

In chapter 3, when we first looked at data binding, we noted that data binding occurs from a source to a target, as in this diagram.

Figure 11.4 A binding occurs from a source object (the binding context) to a target object. The target object (the view) must inherit `BindableObject`, and the target *property* must be a bindable property.



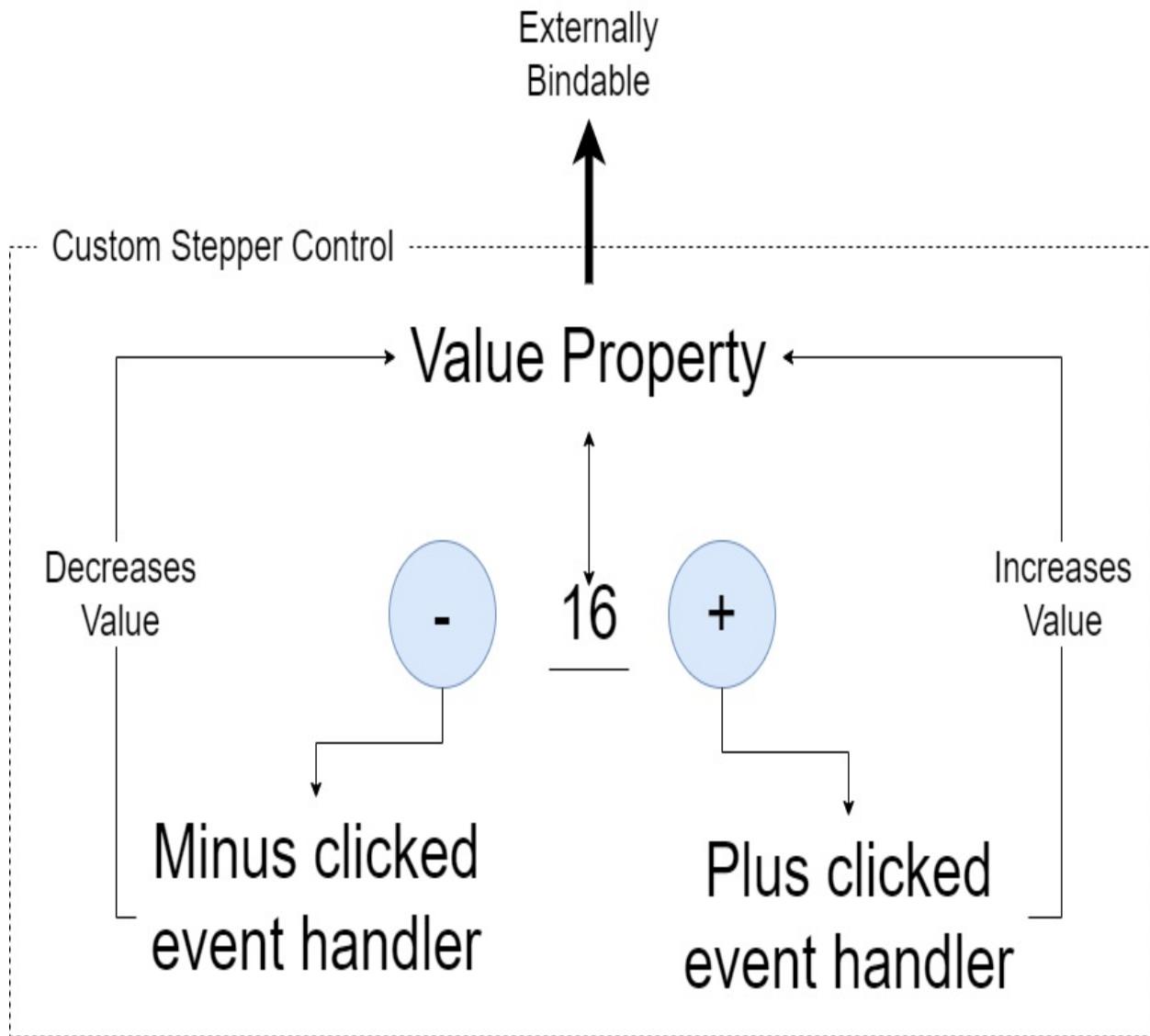
The target of a binding must inherit the `BindableObject` base class. The custom stepper we have created is a `ContentView`, which is a descendant of `BindableObject`. But the target *property* of a binding must be a bindable property (the source of a binding can be a regular property, as we have seen with the ViewModels we've been using so far).

For example, we previously bound the `Value` property of a `Stepper` and the `Text` property of a `Label` (the targets) to properties in a `ViewModel` (the sources). `Stepper.Value` and `Label.Text` are bindable properties. The `Value` property in the `MildredStepper` control is a property, but not a *bindable* property, which means that if we tried to add this to the input page it wouldn't work:

```
<controls:MildredStepper Value="{Binding Count}">
```

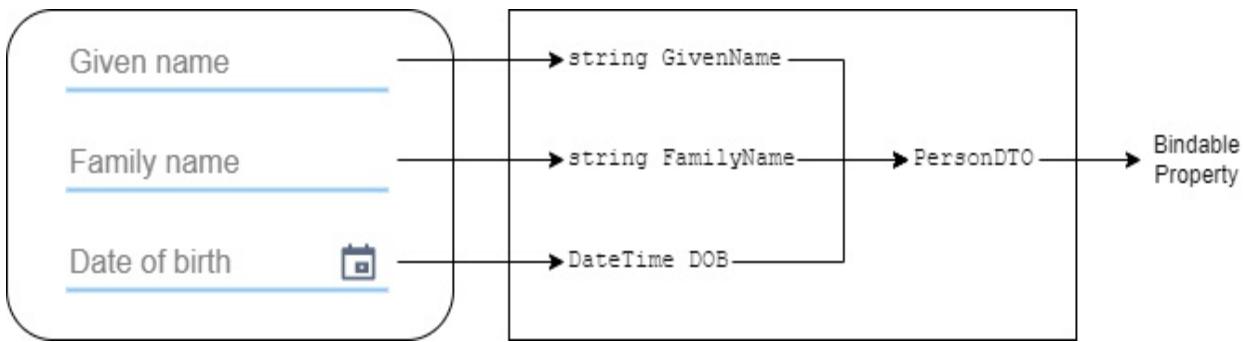
But this is functionality we need, so we need to make `Value` a bindable property. We're going to wrap the inner workings of our control and expose a `value` externally via a bindable property, as shown in figure 11.5.

Figure 11.5 The two Buttons and the Entry that make up the custom stepper control all manipulate or are manipulated by values in the code behind. But a single property is exposed externally as a bindable property, meaning it can be get and set with a binding.



Our custom stepper is a simple example, and while we need to create a bindable property to make the value accessible externally, this ‘wrapping’ can be even more valuable in complex scenarios. Imagine, for example, a templated control that lets you edit details about a person (first and last name, DOB, etc), but exposes a unified person class as a bindable property, as in figure 11.6.

Figure 11.6 A custom control that contains input fields for given name, family name, and date of birth. These directly set values of properties in the code behind, but a single bindable property is exposed that wraps all three properties into a single DTO.



For now, we'll focus on our simpler use case, and create a bindable property for the `Value` property in the custom stepper.

11.2.1 Adding the Value property

Adding bindable properties to a control in .NET MAUI requires following some conventions. The static `Create` method on the `BindableProperty` class is used to create bindable properties, which must be created with the `static` and `readonly` modifiers.

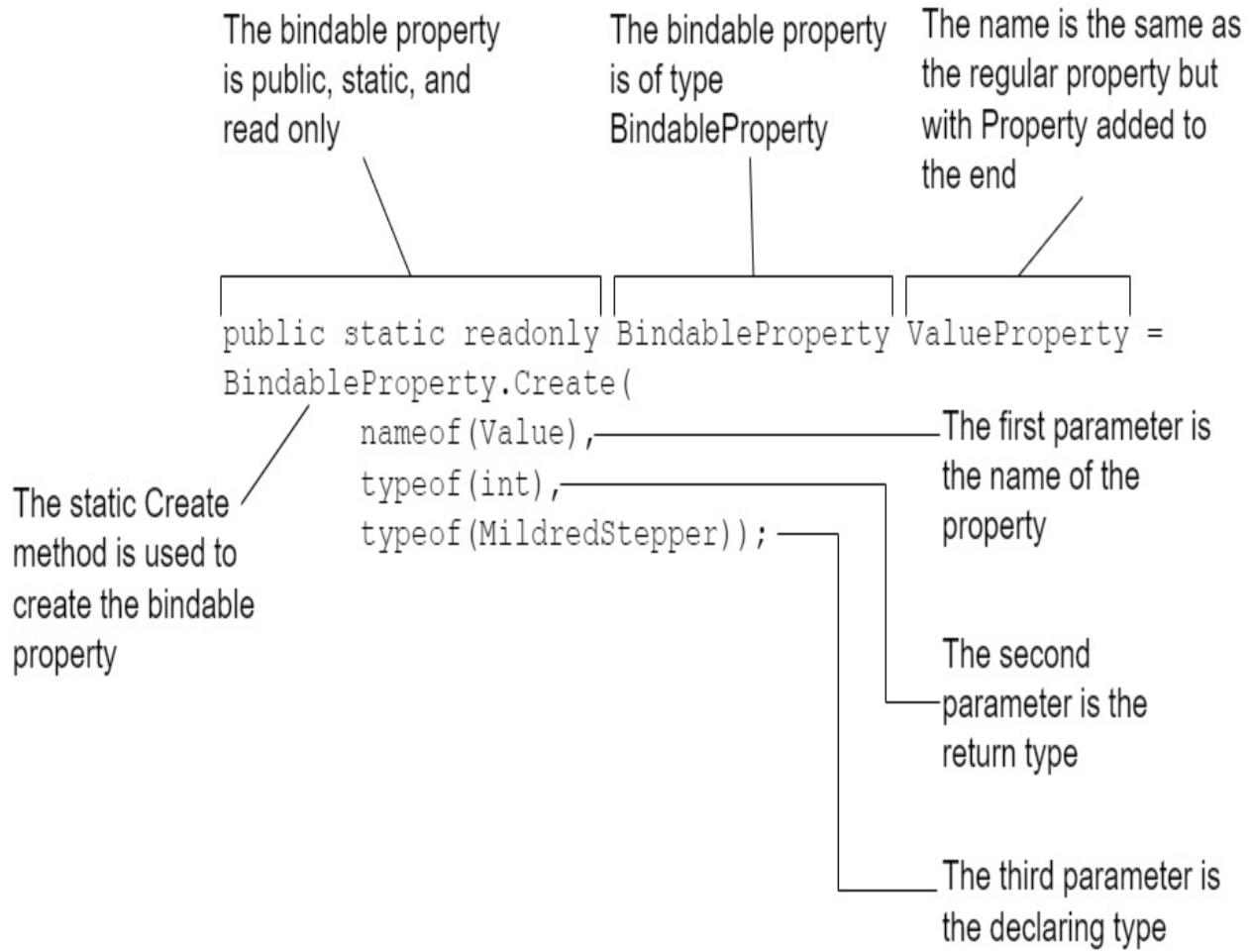
Remember that bindable properties back regular properties (instead of fields). The name of a bindable property must match the name of the property it is backing, with the `Property` suffix. In our custom stepper, we have a property called `Value`, so the bindable property that will back it must be called `ValueProperty`.

The `Create` method has three required parameters:

- The name of the property that it backs
- The return type, i.e., the type of the property that it backs
- The declaring type, i.e., the type of the custom control that the bindable property belongs to

With this information, we can create the bindable property for the `Value` in our custom stepper, using the code shown in figure 11.7.

Figure 11.7 The static `Create` method on the `BindableProperty` type is used to create instances of bindable properties. These must be public, static and read only. The `Create` method requires the name of the property, the type of the property, and the type of the control the property belongs to. The name of the bindable property must match the name of the property, with “`Property`” appended.



We can use this code to make `Value` a bindable property on the custom stepper control, but there's another step we need to make the binding work. The getter and setter of the regular `Value` property need to call the `GetValue` and `SetValue` methods respectively. These methods are inherited from the `BindableObject` base class.

The `GetValue` and `SetValue` methods need to be added to the getter and setter of a property to return and set the value. The `GetValue` method takes a single parameter: the name of the bindable property. `GetValue` returns `object`, so you need to cast the result to the type of your property (`int` in the case of the `Value` property in the custom stepper).

`SetValue` takes two parameters: the name of the bindable property, and the value that you want to assign to the bindable property. With this information in mind, we can add the bindable property and update the `Value` property to work with it.

Listing 11.4 The BindableProperty and updated property

```
public static readonly BindableProperty ValueProperty = BindableP  
    nameof(Value),  
    typeof(int),  
    typeof(MildredStepper));  
  
public int Value  
{  
    get => (int)GetValue(ValueProperty); #A  
    set => SetValue(ValueProperty, value); #B  
}
```

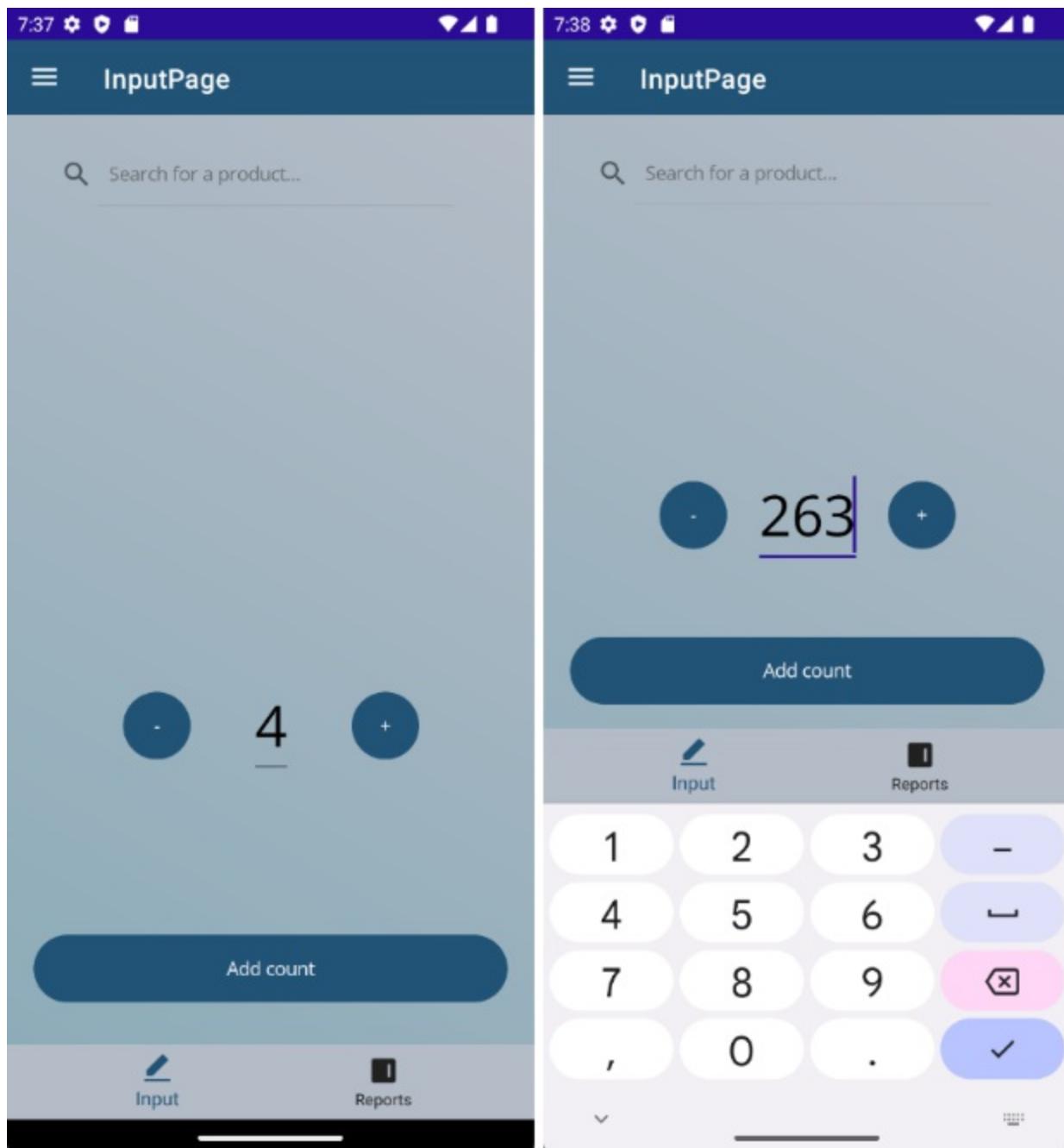
With the bindable property added, the custom stepper is now ready to use. Go back to the input page and add the binding from the `Value` property of the `MildredStepper` to the `Count` property of the page's binding context. Listing 11.5 shows the updated control in the input page.

Listing 11.5 The MildredStepper with the binding added

```
<controls:MildredStepper Grid.Row="2"  
    HorizontalOptions="Center"  
    VerticalOptions="Center"  
    Value="{Binding Count}"/>
```

If you run the app now, you'll see the custom stepper on the page. You can increase and decrease the count using the plus and minus buttons and edit the value directly. As shown in figure 11.8. You should be able to record a stock count and, if you go to the reports page, see that the count is successfully saved with the value from the custom stepper.

Figure 11.8 The InputPage with the custom MildredStepper control added. On the left, the count has been changed by using the plus and minus buttons. On the right, the value has been edited directly, and the numeric keyboard makes it easy for the user to enter the correct type of data.



11.2.2 Adding the IsEnabled property

As you can see in figure 11.8, the custom stepper is enabled by default, and changing the count subsequently enables the add count button, meaning we could submit a count with no product selected and get an error. Before we introduced our custom control, we controlled the `IsEnabled` property of the standard Stepper using a DataTrigger, which would disable the control

when the `SelectedProduct` property of the binding context was null and enable it once a product has been selected. When using custom controls to improve UX, it's important not to compromise any existing functionality, so let's reintroduce this feature.

Adding the bindable property

`MildredStepper` is of type `ContentView`, which already has an `IsEnabled` property inherited from the `VisualElement` base class. We can bind to this from the containing view, and get and set its value this way, but it doesn't currently support the functionality we need for our data trigger. We could add a different property with a different name, but this wouldn't be consistent with the existing convention used by controls in .NET MAUI.

We can work around this by using the `new` keyword. We can add the bindable property as we would any other bindable property, and though it already exists on a base class, we can add the `new` modifier to hide the inherited member. There is also an `IsEnabled` property that the bindable property backs, so we can reuse this and refer to it in the `Create` method, without needing to add it again. Listing 11.6 Shows the new bindable property to add to `MildredStepper.xaml.cs`.

Listing 11.6 The new bindable property for the IsEnabled property

```
public static new readonly BindableProperty IsEnabledProperty = B
    nameof(IsEnabled),                      #B
    typeof(bool),                          #C
    typeof(MildredStepper));               #C
```

So far, the only thing that's changed here is the addition of the `new` keyword, and functionally this bindable property is no different from the inherited one. To bring back the functionality that we need for the data trigger, we need to make a couple of changes:

- **Change the default value.** The default value of `bool` is `false`, but the data trigger we are using needs to *change the value to true*. Remember that data triggers revert their changes when the condition is no longer met, so if the trigger sets `IsEnabled` to `false`, when the

`SelectedProduct` is no longer `null`, it would revert to its default value, which is also `false`. We need it to revert to `true`.

- **Add a change handler.** When the value of the `IsEnabled` property changes, we want to programmatically enable or disable the `Buttons` and `Entry` within our templated control. By adding a change handler, we can inspect the new value and the old value and respond accordingly.

Adding default values

Primitive types in .NET all have a default value (for example, `false` for `bool` or `0` for `int`). With a regular property you can override the type's default and assign a default value to the instance, either when you declare it or in a class constructor, but with bindable properties you need to assign the default value in the `Create` method.

So far, we've specified three parameters for the `Create` method; to specify a default value, we just provide it as the fourth parameter. **Update the bindable property declaration to include a default value of `true`.** Listing 11.7 shows the updated bindable property declaration for the `IsEnabled` bindable property.

Listing 11.7 The bindable property declaration with a default value provided

```
public static new readonly BindableProperty IsEnabledProperty = B
    nameof(IsEnabled),
    typeof(bool),
    typeof(MildredStepper),
    true);      #A
```

With this change, the default value of the bindable property will be `true`, which is what we need to support the data trigger we will add later in this section.

Adding a change handler

The `BindableProperty.Create` method allows you to specify a delegate to be invoked when the value of the property changes, with a parameter called `propertyChanged`. So far we've been using positional arguments in the

Create method, but `PropertyChanged` is not the next argument in the sequence, so we'll need to supply it as a named argument.

Before we add the argument, let's build the delegate, which needs to be a static method with a specific signature. The method will specify three parameters:

- A `BindableObject`, which will be the calling templated control. In this case, the instance of `MildredStepper` that the `BindableProperty` instance belongs to.
- An object which will represent the old value (the value of the property before the change).
- An object which will represent the new value (the value of the property after the change).

These parameters will cover any bindable property of any bindable object, so they need to be cast to the specific types needed for any given property. It's also a good idea to ensure they are the correct type, and for the bindable object, we can do both at once.

Getting both the old value and the new value means you can compare these and act accordingly. But in our case, we're only concerned with the new value. We need to cast it to a `bool` and set the corresponding property on the bindable object to the new value. We don't need to check the value; we can simply set the `IsEnabled` property of the individual controls (the `Entry` and the two `Buttons`) on the bindable object to whatever value we've received.

Listing 11.8 shows the `OnEnabledChanged` method to add to `MildredStepper.xaml.cs`.

Listing 11.8 The `OnEnabledChanged` method

```
private static void OnEnabledChanged(BindableObject bindable, o
{
    if (bindable is MildredStepper mildredStepper) #B
    {
        mildredStepper.IsEnabled = (bool)newValue; #C
        mildredStepper.ValueEntry.IsEnabled = mildredStepper.IsEnabled;
        mildredStepper.PlusButton.IsEnabled = mildredStepper.IsEnabled;
        mildredStepper_MINUSButton.IsEnabled = mildredStepper.IsEnabled;
```

```
    }  
}
```

Now that we've got the method, all that remains is to assign it to the `IsEnabledProperty` in the `Create` method using a named argument. Listing 11.9 shows the updated code for the `IsEnabledProperty` declaration.

Listing 11.9 The `IsEnabledProperty` declaration with the `PropertyChanged` delegate

```
public static new readonly BindableProperty IsEnabledProperty = B  
    nameof(IsEnabled),  
    typeof(bool),  
    typeof(MildredStepper),  
    true,  
    propertyChanged: OnIsEnabledChanged);      #A
```

With that, the `.IsEnabled` bindable property and the `MildredStepper` control are complete. The final step is to add the data trigger back into the input page.

Adding the DataTrigger

The process for adding the data trigger for the custom stepper is the same as it was for the stock Stepper. We define the control's `Triggers` collection and add a `DataTrigger` with a `TargetType`, `Binding` and `Value`. Then we add a `Setter` with a `Property` and a `Value`. The difference is that, as the control is not in the standard XAML namespace, we need to include the XML namespace when defining the `Triggers` collection and the `TargetType`. The binding, value, and setter are identical to what we used with the stock Stepper.

Listing 11.10 shows the updated code for the `MildredStepper` in `InputPage.xaml`, with the data trigger included.

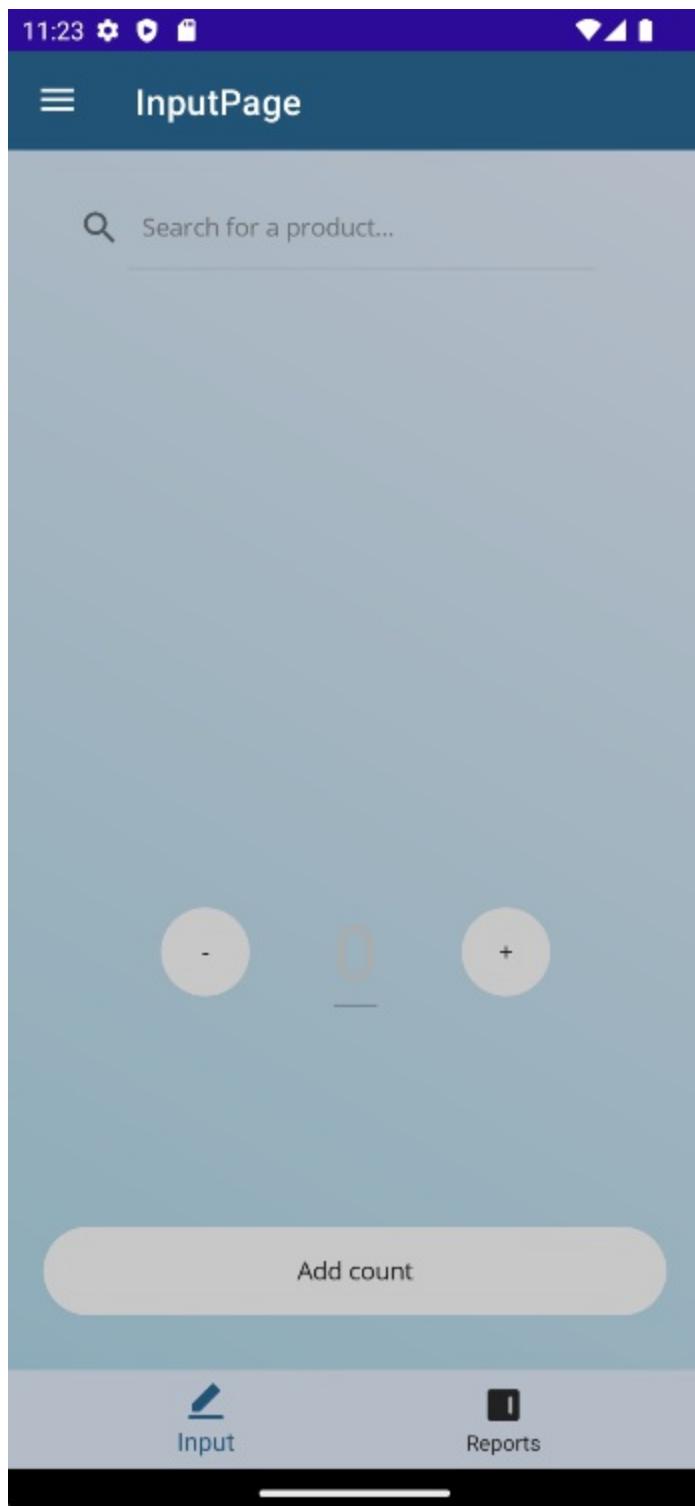
Listing 11.10 MildredStepper with the DataTrigger added

```
<controls:MildredStepper Grid.Row="2"  
    HorizontalOptions="Center"  
    VerticalOptions="Center"  
    Value="{Binding Count, Mode=TwoWay}">  
    <controls:MildredStepper.Triggers>          #A
```

```
<DataTrigger TargetType="controls:MildredStepper"
    Binding="{Binding SelectedProduct, TargetNullValue=""}>
    <Setter Property="IsEnabled"
        Value="False" />
</DataTrigger>
</controls:MildredStepper.Triggers>
</controls:MildredStepper>
```

This completes the work for the custom stepper control. If you run the app now, you should see the custom stepper disabled, as in figure 11.9, until you search for and select a product. Once you do this, the stepper will be enabled, and you can set a count, and subsequently submit it.

Figure 11.9 The custom stepper is disabled by a DataTrigger when the SelectedProduct in the binding context is null. Searching for and selecting a product enables the custom stepper.



An easier way to create bindable properties

In previous chapters we saw how the `INotifyPropertyChanged` interface is

used to notify the UI that properties have changed in their binding context. This is more complex than in some other UI frameworks, but the process can be simplified with source generators in the MVVM Community Toolkit. Instead of writing the property and the field and invoking the `PropertyChanged` event in the setter, you simply declare the field and decorate it with an attribute. To find out more about this awesome feature, check out this video from James Montemagno:
<https://youtu.be/aCxl0z04BN8>.

Creating bindable properties is significantly more laborious and, unfortunately, the .NET MAUI Community Toolkit doesn't include such a source generator for bindable properties (although at time of writing there is an open proposal and spec for one). However, there is a package available that does exactly this. You can find out more about it here:
<https://github.com/rrmanzano/maui-bindableproperty-generator>.

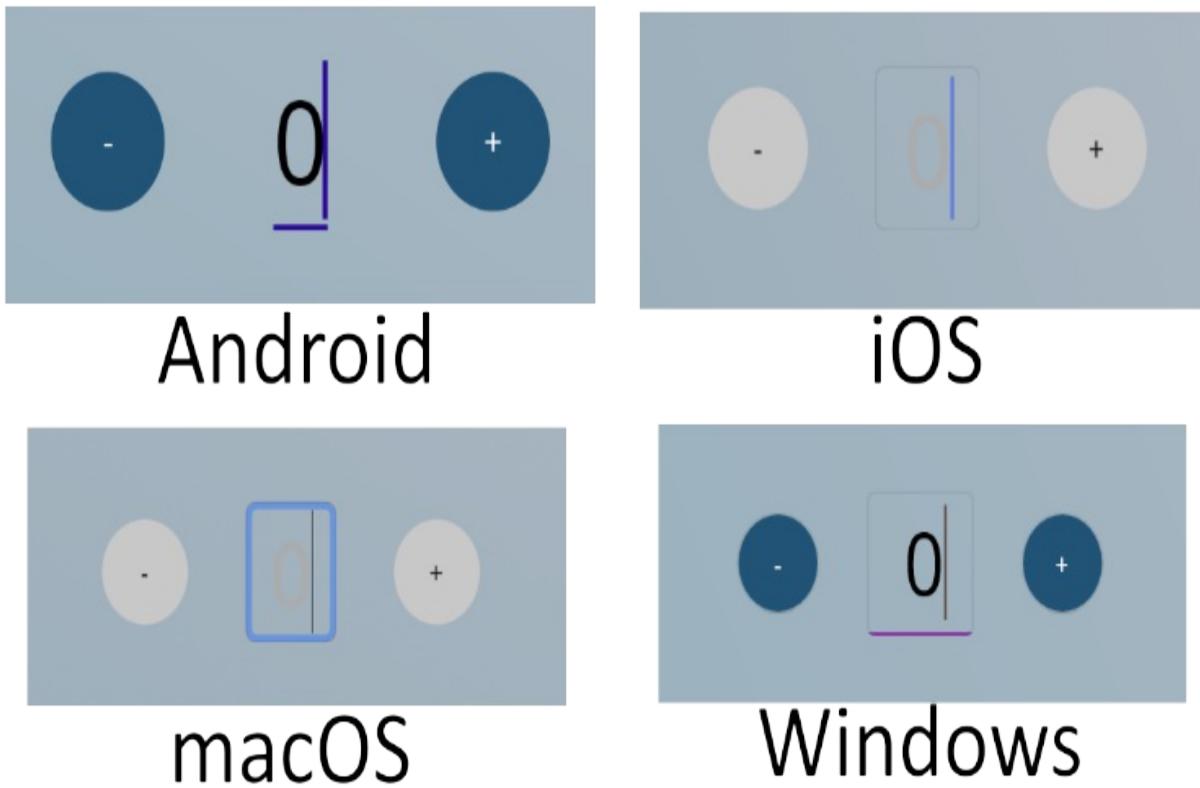
I recommend that you continue to create bindable properties manually until you can do so without referring to this book or the documentation, to ensure that you have a thorough understanding of how they work. But once you do, switching to this package could be a significant time saver.

11.3 Modifying platform controls with handlers

The new custom stepper control we have added provides a significant UX improvement, especially for inputting large numbers. After a round of testing, Mildred's staff have found that they prefer the functionality over the standard Stepper, but the UI is unpopular. Mildred asked her designers to suggest improvements, and they have asked you to make the `Entry` in the middle of the stepper a bit less conspicuous.

Figure 11.10 compares how the custom stepper currently looks across the supported platforms. Each platform takes a slightly different approach to how it renders an `Entry`, and therefore how the stepper is rendered.

Figure 11.10 The custom stepper is similar on each platform, but with some variations. On all platforms the Entry is prominent which makes it discoverable, but unsightly, particularly when just the buttons are used.



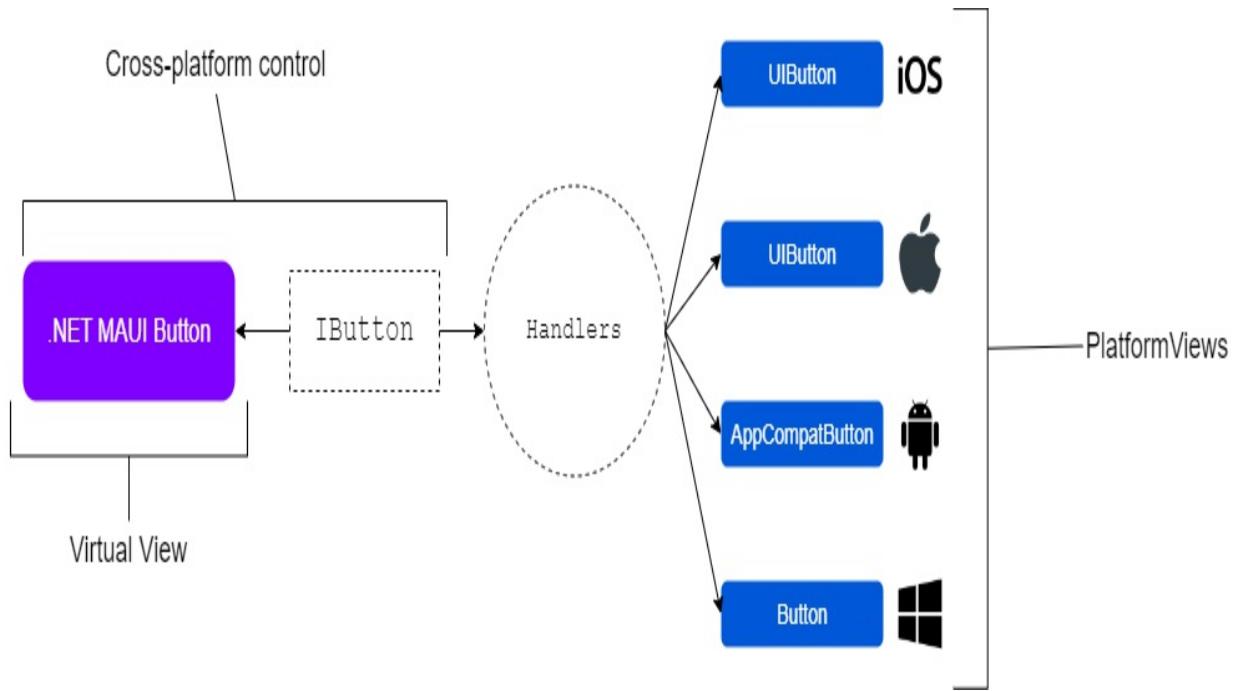
In the last chapter, we saw how we can use styles and control properties to change the appearance of out of the box controls; but there is no property exposed by .NET MAUI to control the border of an `Entry` control. When we encounter situations like this, we can override the way that the .NET MAUI control abstraction is implemented on the target platforms, and we do this by customizing the control's *handler*.

11.3.1 Handler architecture

The cross-platform controls that we have been using in our .NET MAUI apps are represented by abstractions, meaning that each is essentially a conceptual definition of a UI control. *Virtual views* implement these abstractions as the controls we use in .NET MAUI apps, and *handlers* map the abstractions to *native views*, their specific implementations on each platform. Each handler has a property called `PlatformView` that represents the native control. Handlers are the glue that binds together the cross-platform controls to platform-specific implementations and are the core technology that lets us write cross-platform apps in .NET MAUI. Figure 11.11 shows the

architecture of handlers in .NET MAUI.

Figure 11.11 Cross-platform controls are described by interfaces, and virtual views implement them as controls in the .NET MAUI UI layer. Handlers map the abstractions to platform-specific implementations.



Handlers define *mappings* in a dictionary that describe how these cross-platform properties are applied on each platform. For example, a `Button` in .NET MAUI has several properties that we can modify, including `BackgroundColor`. A handler maps the .NET MAUI `BackgroundColor` property, which is of type `Microsoft.Maui.Graphics.Color` to the platform-specific property, which, on iOS and macOS for example, is of type `UIKit.UIColor`.

NOTE

One handler exists for each control and each platform. For example, there is one `Button` handler for Android, one for iOS, one for Windows and one for Mac Catalyst.

Each platform implements UI controls differently, but when building .NET MAUI apps, we're not usually concerned about the platform-specific

implementation details. Sometimes, though, to get a finer-grained level of control over how UI elements are displayed, we need to override them.

We can also create our own handlers to create our own cross-platform controls, to gain access to platform-specific controls that have not been exposed in .NET MAUI. If you find a platform-specific control that you need access to in your .NET MAUI app, you can read more about this approach in the documentation here: <https://learn.microsoft.com/dotnet/maui/user-interface/handlers/create>. For our custom stepper, we will only need to modify existing mappings.

11.3.2 Overriding handler mappings

When we need to modify the appearance of a control beyond what is provided by the cross-platform abstraction, we can override the handler mappings. Each handler has a mapper, and each mapper provides three methods for overriding the mappings:

- `PrependToMapping`: The changes you specify in here are applied *before* the default mappings in the handler. This can be a good option if you're adding mappings that aren't currently defined.
- `ModifyMapping`: This changes existing mappings defined in the handler.
- `AppendToMapping`: The changes you specify in here are applied *after* the default mappings in the handler. This means that changes here will take precedence over the defaults.

When do I use each method?

The `PrependToMapping` method can be useful if you want to map properties that aren't already mapped by the default handler mappings, but you *don't want your mappings to override anything in the defaults*.

`ModifyMapping` can be useful if you have a deep understanding of the existing mapping dictionary and want to change the way the default mappings are defined.

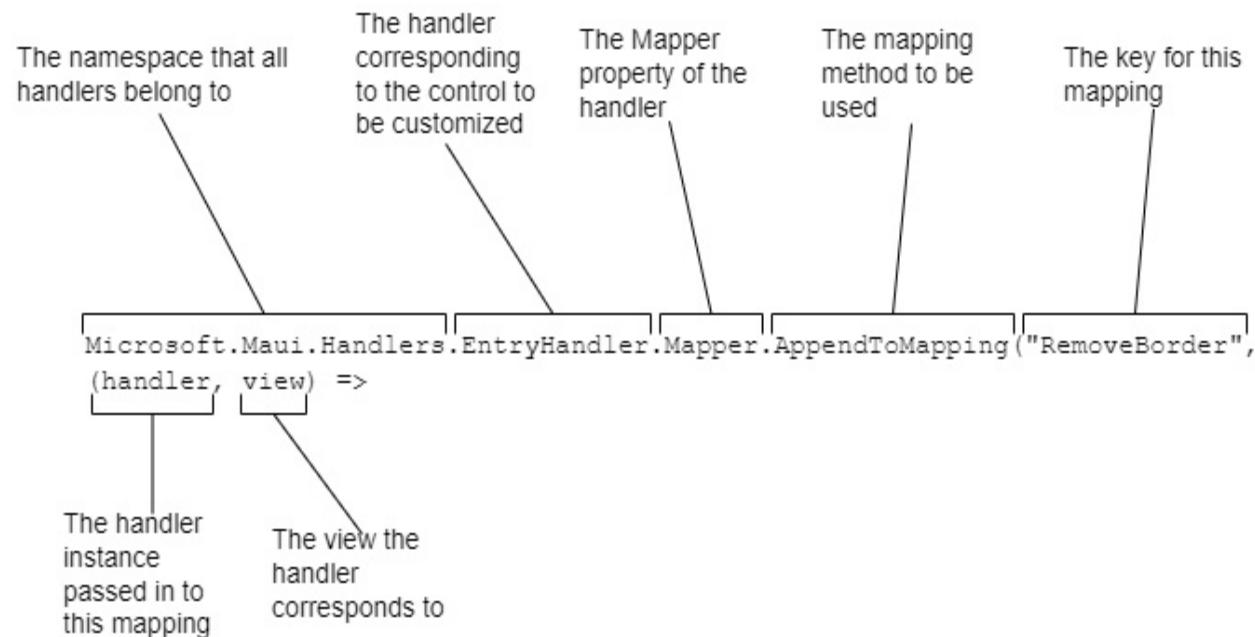
`AppendToMapping` gives you the most assurance that your customizations will be applied and is the method you should use in almost all cases. It's unlikely

that you'll need to use one of the other methods in most cases.

All three methods have the same two parameters. The first is a key; mappings are defined in dictionaries, so if you're using the `ModifyMapping` method, you must use the key for the default mapping (in .NET MAUI these are defined as the name of the property on the relevant interface). For the other two methods, you can use whatever `string` key you like.

The second parameter is an `Action`, which has two arguments that get passed in for you. The first is the handler that the mapper belongs to (i.e., the platform-specific instance of the handler), the second is the view that the handler corresponds to. Figure 11.12 shows a sample of adding a handler mapping to a view.

Figure 11.12 This code opens a lambda expression for appending mappings to the handler for the cross-platform `Entry` control. The mappings for this control to platform-specific controls are in the `EntryHandler`, which is in the `Microsoft.Maui.Handlers` namespace. The `AppendToMapping` method has been used, which is on the `Mapper` property of the handler. The key for this mapping profile is `RemoveBorder`, but if this were using the `ModifyMapping` method instead, an existing key in the mapping dictionary would need to be used. The Action gets two arguments passed in: the handler instance, and the view being customised.



There are a few places you can implement your handler mappings. One common approach is to put them in `MauiProgram` as part of the app's startup

logic; another is to put the handler logic in the relevant platform folders. A third approach, as we will use later in this chapter, is to keep the handler mapping with the control that we are modifying.

An important thing to remember is that, once executed, your handler modifications will apply to *all* instances of the control throughout the app, and where you put your handler logic determines when that logic gets executed. For example, if you put it in `MauiProgram`, it will be executed before any views are rendered, and the modifications will apply to all instances of the view as soon as the app starts. If you put it somewhere else, it will be executed once that code path is reached; at which point *all* instances of the control the handler is responsible for will be modified.

If you don't want to apply a modification to all instances of a control, you can subclass it and check within your handler logic whether the affected view is an instance of the base class or of your subclass. This is the approach we will take later in this chapter.

With a subclassed control, my preferred approach is to keep the handler mappings in the constructor. This ensures that any time an instance of my control is rendered, the handler logic will be executed, and it keeps all the rendering logic for the custom control in one place. If I wanted to alter all instances of a control, I would put the handler mappings in `MauiProgram`, and if I wanted to modify a control on one platform only, I would put the handler logic in that platform folder.

Let's see how we can put this into action to make the modifications Mildred is requesting for her app.

11.3.3 Implementing custom handler logic

For the custom stepper, we need to modify the `Entry` control, but the mapping dictionary in the `EntryHandler` applies to *all* instances of `Entry`. Once we apply the mappings, they will apply across the board, which isn't what we want. Instead, let's subclass the `Entry` type and in our `AppendToMapping` method, we'll check to make sure we're only applying it to the desired type.

Create a class called `BorderlessEntry` in the `Controls` folder of `MauiStockTake.UI` that subclasses `Entry`. Add a private void method called `ModifyEntry` and call it from the constructor. Inside the `ModifyEntry` method, call the `AppendToMapping` method on the `EntryHandler`'s `Mapper`, and give it a key of `RemoveBorder`. Listing 11.11 shows the boilerplate code for the `BorderlessEntry`.

Listing 11.11 The `BorderlessEntry` class

```
namespace MauiStockTake.UI.Controls;
public class BorderlessEntry : Entry      #A
{
    public BorderlessEntry()
    {
        ModifyEntry();          #B
    }

    private void ModifyEntry()
    {
        Microsoft.Maui.Handlers.EntryHandler.Mapper.AppendToMapping(
            {
            });
    }
}
```

At this point, we could start adding our customizations, however, as we're applying these to the `EntryHandler`, as soon as an instance of `BorderlessEntry` is constructed, these customizations will apply to everything the `EntryHandler` is responsible for, in other words, every instance of the `Entry` control.

Instead, let's add a conditional check inside the mapping logic to ensure we're only applying this to instances of `BorderlessEntry`, and not any instance of `Entry`. Listing 11.12 shows the check to add inside the mapping code.

Listing 11.12 The conditional check in the mapping code

```
Microsoft.Maui.Handlers.EntryHandler.Mapper.AppendToMapping("Remo
{
    if (view is BorderlessEntry)          #A
    {
```

```
}); }
```

The last thing that we need to do before we start applying our customizations is to add some compiler directives so that we can separate the logic for each instance of the handler (i.e., each platform-specific handler). There are other ways we could have done this, for example we could have used partial classes, and added the modifications inside the platform holders. But this approach lets us keep all the logic for each custom control in one place. Listing 11.13 shows the compiler directives to add inside the `if` conditional block.

Listing 11.13 The compiler directives to isolate platform logic

```
#if ANDROID          #A
#elif WINDOWS        #B
#elif IOS || MACCATALYST    #C
#endif
```

The `BorderlessEntry` is now ready for us to apply our platform-specific customizations.

Android

As we saw above, the lambda expression gets two arguments passed in. One is the view being customized, and the other is the platform-specific handler instance. The handler instance has a property called `PlatformView` that gives us access to the native control that the handler maps to. In the case of Android, this is an `AppCompatEditText` widget.

To get rid of the borders and the underline on Android, all we need to do is set the `Background` property of the `AppCompatEditText` widget to `null`, and call its `SetBackgroundColor` method, passing in `Android.Graphics.Color.Transparent` as an argument. We use the fully qualified name for the color because we've already got compiler directives inside the mapper; if we start adding them to the `using` statements too, the code will get messy.

Listing 11.14 shows the two lines to add inside the `ANDROID` section of the

compiler directive `#if` block.

Listing 11.14 The Android specific customizations

```
handler.PlatformView.Background = null;           #A  
handler.PlatformView.SetBackgroundColor(Android.Graphics.Color.Tr
```

How do I know what changes to make to platform-specific mappings?

Working with handlers is a breeze, especially compared to the renderers architecture it replaces in Xamarin.Forms. The difficulty comes from knowing which native controls are being mapped to and what properties on them to change.

This is a skill you will develop as you progress as a .NET MAUI developer, particularly as you learn more about the target platforms. You may find that eventually you have your own library of platform customizations that you curate, but there are a few ways you can figure out what changes you need to make to mappings.

A good way to explore this is with IntelliSense. If you're using an IDE like Visual Studio, IntelliSense will tell you which properties and methods are available. Sometimes it will be obvious what kinds of values you can assign to them, other times you can look them up in (say) the Android developer documentation. You can also hover your mouse over `PlatformView` or its properties in the editor to see which native control is in use; you can then look up what its properties are, or even find documentation or guides on how to perform the specific customization you're trying to achieve. Then all you need to do is translate it into a mapping in the handler; but this is usually easier than the first part.

However, it's likely that most of the modifications you will need in your apps are already well-documented by the community, and even if you can't find something .NET MAUI specific, you'll almost certainly find it for Xamarin.Forms. While Xamarin.Forms uses a different architecture, translating renderers to handlers is usually a simple task. In fact, there's an example of translating the `BorderlessEntry` we're creating here from a Xamarin.Forms renderer to a .NET MAUI handler linked in appendix B.

The .NET MAUI community is one of the best things about .NET MAUI development. It's an active and vibrant community full of people who love sharing knowledge and supporting their peers. When you need to customize a handler, your favorite search engine will almost certainly turn up a relevant blog post, video, or discussion. And if not, you can always reach out to the .NET MAUI community using the links in appendix A.

Eventually, you'll likely start figuring these out for yourself, using a combination of IntelliSense, platform documentation, and your growing skillset. When you do, consider documenting your findings. Blog posts and videos are popular, and there are regular community showcase events where people share these kinds of things. The community will thank you, and you may also thank yourself when you come back and refer to it later!

This removed the `Entry` chrome, which looks great when using the stepper buttons. But there are none of the usual UX cues a user expects when editing text fields. Before we make the changes to the other platforms, let's add a `Border` around the `Entry` in the custom stepper. We'll set the `StrokeThickness` to 0 so that it's not usually visible and use a data trigger to show it when the `Entry` has focus (i.e., when a user taps on or clicks into it). We can use the `OnPlatform` markup extension so that it only shows on Android. Listing 11.15 shows the changes to the `MildredStepper.xaml` file to add the border.

Listing 11.15 The conditional border to add to `MildredStepper.xaml`

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentView ...>
    <Grid ...>
        ...
        <Border Grid.Column="1"
            Stroke="{StaticResource Primary}"
            BackgroundColor="Transparent"
            Margin="10"
            StrokeThickness="0"
            StrokeShape="RoundRectangle 5">
            <Border.Triggers>
                <DataTrigger TargetType="Border"
                    Binding="{Binding Source={x:Reference
                        Value="True">
                <Setter Property="StrokeThickness"
```

```

        Value="{OnPlatform Android=1}"/>
    </DataTrigger>
</Border.Triggers>
<Entry x:Name="ValueEntry" .../>
</Border>
...
</Grid>
</ContentView>
```

This completes the modifications necessary for Android, let's move on to iOS and macOS.

macOS and iOS

Back in chapter 3 we saw that .NET MAUI apps run on macOS through Catalyst, which runs iOS apps on macOS, while still giving access to macOS features when needed. Because of this, the customizations for iOS and macOS are the same, and we can put them into the same conditional compiler block.

On iOS and macOS, the `PlatformView` is a `UITextView`. We'll set its `BackgroundColor` property to `UIColor.Clear` in the `UIKit` namespace, and its `BorderStyle` property to `UITextBorderStyle.None`, also in the `UIKit` namespace. We also need to set the `BorderWidth` property, from the `UIView` base class that `UITextView` inherits, to 0. Listing 11.16 shows the code to add to the macOS and iOS conditional compiler block.

Listing 11.16 The macOS and iOS customizations

```
handler.PlatformView.BackgroundColor = UIKit.UIColor.Clear;
handler.PlatformView.Layer.BorderWidth = 0;
handler.PlatformView.BorderStyle = UIKit.UITextBorderStyle.None;
```

This completes the changes we need to make for iOS and macOS, so let's move on to Windows.

Windows

The `PlatformView` on Windows is a `TextBox` in the

`Micorosft.UI.Xaml.Controls` namespace. It has a `Background` property which we need to set to `null`, and a `BorderThickness` and `FocusVisualMargin` property which we need to set to a new instance of `Microsoft.UI.Xaml.Thickness`. This takes an `int` in its constructor that defines its thickness value, so we'll pass in `0`. Listing 11.17 shows the code to add to the Windows conditional compiler block.

Listing 11.17 The Windows customizations

```
handler.PlatformView.BorderThickness = new Microsoft.UI.Xaml.Thic  
handler.PlatformView.Background = null;  
handler.PlatformView.FocusVisualMargin = new Microsoft.UI.Xaml.Th
```

The `BorderlessEntry` class is now complete, and while it removes the border from the `Entry` on Windows, the focus underline will still appear. We can't remove this using a handler; instead, we need to modify the resource dictionary in the `App.xaml` file in the Windows platform folder. The Windows `App.xaml` file contains a root node of `maui:MauiWinUIApplication`. Add the code in listing 11.18 to the `App.xaml` file between these tags.

Listing 11.18 The code to add to App.xaml in the Windows platform folder

```
<maui:MauiWinUIApplication.Resources>  
    <Thickness x:Key="TextControlBorderThemeThickness">0</Thickne  
    <Thickness x:Key="TextControlBorderThemeThicknessFocused">0</  
</maui:MauiWinUIApplication.Resources>
```

These properties are not exposed via the `PlatformView`, so this is the only place we can make this change. This introduces a new problem, though, which is that as we are not applying this to a specific subclass, *every* `Entry` will now be completely borderless on Windows, and will lose the focused underline. While we're only using the `Entry` control as part of other controls in `MauiStockTake`, on Windows the `SearchBar` (which we have at the top of the input page) uses the `TextBox` control, which this change applies to, under the hood, so will be affected by this change.

We could resolve this by subclassing the WinUI `TextBox` control in the Windows platform folder; then we could apply this Windows-specific style change to the subclass rather than the base `TextBox`. Finally, we could create

an entirely new mapping (rather than modifying the existing one) that maps our subclassed `BorderlessEntry` to our subclassed `TextBox` derivative.

However, as we're not using `Entry` anywhere else, this is not necessary for `MauiStockTake`, and an easier approach is to just fix up the `SearchBar` for now. Let's add a `Border` around it but use the `OnPlatform` markup extension to give the `Border` a `StrokeThickness` of `0` on all the other platforms.

Making tradeoffs

In `MauiStockTake`, we don't have any explicit instances of the `Entry` control, so sacrificing the borders and underline isn't a huge issue. We can accept that compromise to achieve the desired effect with a control that we *do use*.

As the app grows, we will likely find we will use the `Entry` in other places. When this happens, we have a choice to make: we can either revert this change on Windows, and accept that the custom stepper doesn't 100% match the design (but maybe it's close enough), in exchange for using the `Entry` control as it is provided out of the box, or we can create a custom entry control, for example introducing our own border or underline focus effects, which we would then use everywhere in the app instead of the standard control. I made a video showing this approach with a Material style custom entry control, available free (with sign-up) on Manning's liveVideo platform here: https://livevideo.manning.com/module/1349_1_1/building-a-material-text-entry-in-.net-maui/new-unit/building-a-material-text-entry-in--net-maui.

On larger apps, this latter approach is likely what you'll be doing anyway. Many companies have their own specific branding and style guides and want all their controls, across all their apps and platforms, to look a certain way. In .NET MAUI apps, you may be creating controls to achieve this and sharing them via control libraries, as we mentioned in chapter 8 and will revisit later in this chapter.

Listing 11.19 shows the border code to add to `InputPage.xaml`.

Listing 11.19 The Border workaround for SearchBar in InputPage.xaml

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

```

<ContentPage ...>
    <Grid ..>
        <Border StrokeShape="RoundRectangle 5"
                Stroke="{StaticResource Primary}"
                BackgroundColor="Transparent"
                StrokeThickness="{OnPlatform WinUI=1, Default=0}"
                <SearchBar .../>
            </Border>
            ...
        </Grid>
    </ContentPage>

```

This completes the borderless entry control; all we need to do now is drop it into the custom stepper.

11.3.4 Updating the custom stepper

The custom stepper control has an `Entry` in the central column of its `Grid`. Because the `BorderlessEntry` is a subclass of `Entry`, we can just drop it in as a direct replacement (recall the Liskov Substitution Principle mentioned in the sidebar *MVVM for SOLID apps* in chapter 9). The only thing we need to do first is add an XML namespace in the XAML so that we can refer to the custom control, then we can just do a direct replacement, referencing the `BorderlessEntry` via the XAML namespace.

Listing 11.20 shows code changes for the `MildredStepper.xaml` file.

Listing 11.20 `MildredStepper.xaml` updated to use the new `BorderlessEntry`

```

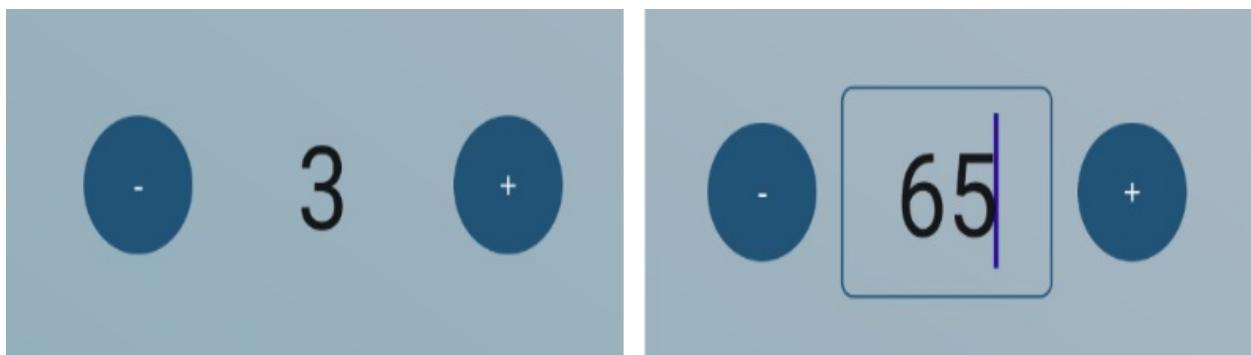
<?xml version="1.0" encoding="utf-8" ?>
<ContentView xmlns:controls="clr-namespace:MauiStockTake.UI.Contr
    <Grid ...>
        ...
        <controls:BorderlessEntry                      #B
            x:Name="ValueEntry"
            Keyboard="Numeric"
            FontSize="42"
            HorizontalTextAlignment="Center"
            TextChanged="ValueEntry_TextChanged"
            VerticalOptions="Center"
            HorizontalOptions="Center"/>
        ...
    </Grid>

```

```
</ContentView>
```

With that final change, the custom stepper is now complete. Run the app, and you should see the updated `MildredStepper` on the input page, as in figure 11.13.

Figure 11.13 The custom stepper with the `BorderlessEntry`, shown here on Android. On the left the number has been changed using the stepper buttons. On the right, the number is being edited directly, and a border is shown.



On Windows, there is a hover effect (called `PointerOver` in the `VisualStateManager`, as we saw in chapter 10), and a focus effect. We could remove these too; however, they lend themselves well to the discoverability of the control, so we'll keep them in (and lament their absence on the other platforms!).

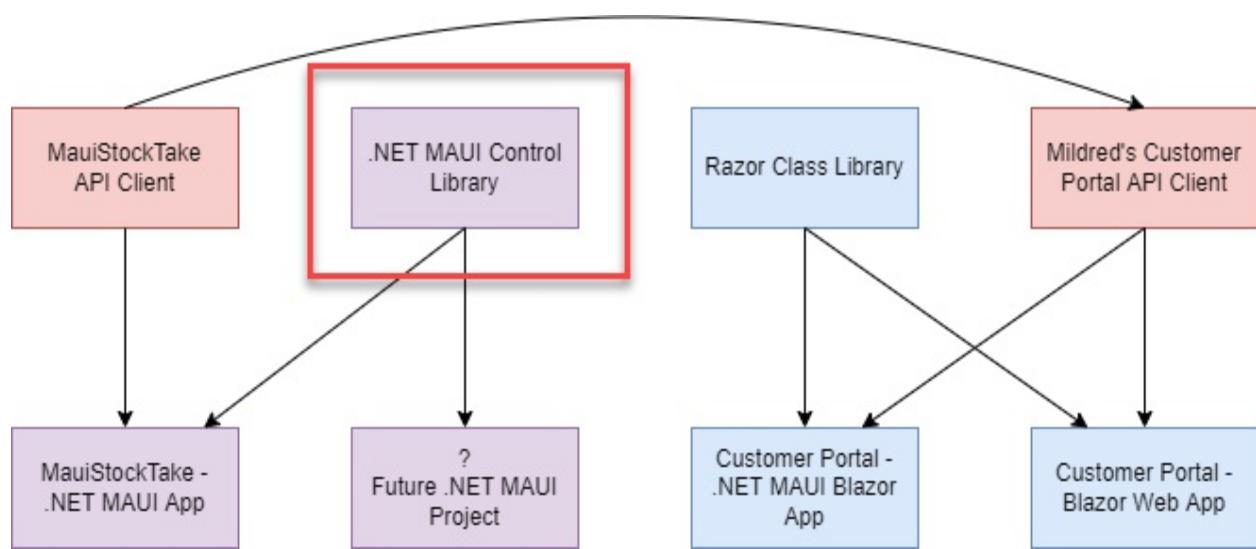
11.4 Creating and sharing control libraries

You can create custom controls using templated controls, by drawing controls with the `Microsoft.Maui.Graphics` library, by customizing native controls with handler mappings, or combinations of all of them. Often this will be to conform to a company's branding or style guide, or even their design language system (DLS). When this is the case, it's useful to be able to reuse these controls across multiple apps, and even when not, sometimes you develop a useful control (or set of controls) that you think is worth reusing.

In chapter 8 we looked at sharing code within a solution, but we also mentioned sharing code across an enterprise. If you're building .NET MAUI apps, it makes sense to share reusable controls in control libraries. For

example, if Mildred's Surf Shack wanted to introduce some more apps, it would make sense to move the custom stepper into a control library that can be shared with other apps. Figure 11.14 shows an updated version of the code sharing diagram from chapter 8 to highlight this.

Figure 11.14 Components of the enterprise app ecosystem at Mildred's Surf Shack. The .NET MAUI Control Library is highlighted; it would make sense to move the custom stepper (and any other custom controls) into here so that it can be used by the stock taking app as well as any other apps in the enterprise.



Naming controls

It's common to see controls with names like `CustomStepper`. This isn't particularly descriptive, and it's better to use more meaningful names. For example, it's easy to tell just from the name what `BorderlessEntry` is and what it does.

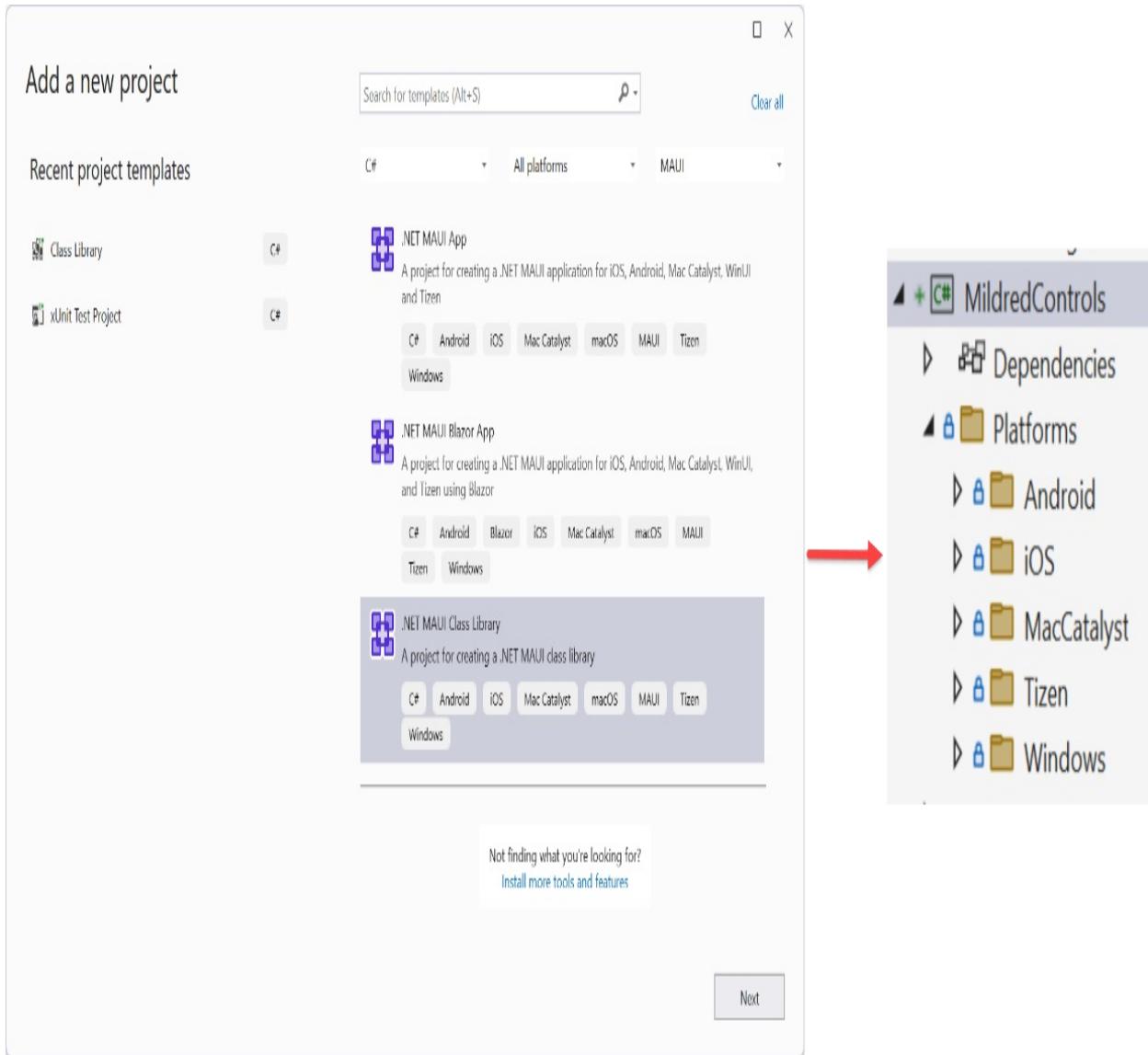
It's also common for developers to name controls after themselves or their company, for example `GoldieEntry` or `SSWButton`. This is ok when you're building and curating (or even sharing) your own control library, but if you're building apps for or on behalf of another business, it's better to name the controls in a way that is meaningful for them, as in the example of the `MildredStepper`.

`MildredStepper` may not provide information about its specific customizations, but it does tell you that it's a variation of a `Stepper` control

that fits in with Mildred's brand. This is a better name than `CustomStepper`, as it's more specific and therefore less likely to clash with controls in other libraries. It's also better than an app-specific name like `StockTakeStepper`, especially useful if it gets moved to a control library and used in more than one app at Mildred's Surf Shack.

Sharing controls with .NET MAUI is straightforward. If we wanted to share the `MildredStepper` control with other apps, the first thing we would do is create a .NET MAUI class library. There is a .NET template for this that you can use from Visual Studio or the .NET CLI, which is like a regular class library but with all the .NET MAUI dependencies already wired up. Figure 11.15 shows the new project dialog for the .NET MAUI class library project template.

Figure 11.15 A .NET MAUI class library with all .NET MAUI dependencies wired up. It includes the platform folders where you can keep platform-specific code.



We could then move (or recreate) the `MildredStepper` control to this library, then add a dependency on this library in the `MauiStockTake` app. We'd have to update the namespace, but otherwise there would be no difference.

.NET makes it easy to share class libraries across an enterprise too. The whole library can be packaged with NuGet, which is configurable as part of the build, as shown in figure 11.16.

Figure 11.16 Creating a NuGet package from a class library, including a .NET MAUI class library, can be configured as part of your build process.

MildredControls X

Search properties

Application Global Usings Build Package

General

Generate NuGet package on build

Produce a package file during build operations.

General

License Symbols

Code Analysis

Package ID ⓘ

The case-insensitive package identifier, which must be unique across nuget.org or whatever gallery the package resides in. IDs may not contain spaces or characters that are not valid for a URL, and generally follow .NET namespace rules.

\$(AssemblyName)

Resources

MildredControls

MAUI Shared

Title

A human-friendly title of the package, typically used in UI displays as on nuget.org and the Package Manager in Visual Studio.

iOS

Once the project has built, you can copy it to a shared folder, which can be added as a NuGet source. A better approach, though, is to have the NuGet package created and distributed as part of your CI/CD pipelines. If it's a control library that you're sharing publicly, the best place is nuget.org. In other scenarios, GitHub can host both private and public NuGet feeds, as can Azure DevOps, and several other for-purpose NuGet solutions.

11.5 Summary

- Templatized controls in .NET MAUI let you ‘componentize’ views. Templatized controls are created using the `ContentView` template.
- Custom controls should always improve UX. If you remove or override some functionality, you should either re-add it, or replace it with something better.
- Validation, and particularly meaningful feedback about failed inputs, will significantly improve your app’s UX.
- Templatized controls use bindable properties to enable data binding between their containing view and their internal data.
- Bindable properties are created using the static `BindableProperty.Create` method. The properties that they back must use the `GetValue` and `SetValue` methods in their getter and setter to get and set values of the corresponding bindable property.
- Bindable properties must follow the naming convention of ‘[property that they back]Property’, for example `IsEnabledProperty` backs a class property called `.IsEnabled`.
- .NET MAUI maps cross-platform controls to native controls using handlers. Handlers contain dictionaries of property mappings that map the properties of cross-platform controls to the native platform controls.
- .NET MAUI class libraries are an awesome way of sharing controls between apps in an enterprise or with the community.

12 Deploying Apps to Production with GitHub Actions

This chapter covers

- Preparing your apps to be used by other people
- Providing your own icon and splash screen
- The Apple, Google and Microsoft developer programs
- Continuous Integration and Continuous Delivery with GitHub Actions

Congratulations! You have built a complete, non-trivial app in .NET MAUI. Building apps is fun, but sadly very few people get paid to build apps just for fun; usually, there's an expectation that you will deliver a working app to users.

We've come a long way since the start of this book. We began with *Aloha, World!*, we've built a location sharing app, a to-do app, a movie recommender, and now an enterprise stocktaking app. Getting to where we are now has been a long journey, and the last, and arguably most important, step is to get our app deployed.

Once you've got a build of your app that's ready to deploy to your users, there are a few final pieces of polish you need, outside of the app's functionality, before you have a finished product.

Once all the pieces are in place, the last step is deployment. As a professional app developer, you will need a CI/CD pipeline and automated build and deployment; as part of an enterprise app development team, they are an essential component of delivering quality software.

In this chapter, we'll see how we can add those last pieces of polish to our app, and as an enterprise software development team at Beach Bytes, we'll use GitHub Actions to create repeatable, reliable builds that we can deliver to the various stores through automation.

12.1 App icons, splash screens, and app identifiers

The icon is the first part of your app that anyone will ever see. The various store listings will feature the icon prominently, and once it's installed onto a device, users will see the icon before even launching the app. Having a unique icon stands out both in the store and on the device, helping users find and launch your app quickly, so it's important to replace the default .NET icon. Mildred's design team have given us an icon to use which you can see in figure 12.1.

Figure 12.1 An app icon for MauiStockTake that's aligned with the surfing theme of Mildred's business



This icon has been provided as two SVG files: `icon-bg.svg` and `icon-fg.svg`. Both of these files can be found in the chapter 12 resources folder. We don't need to worry about merging these, as .NET MAUI composes an app icon for you based on a foreground file and a background file, as shown in figure 12.2.

Figure 12.2 An app icon can be composed of a foreground image layered on top of a background.



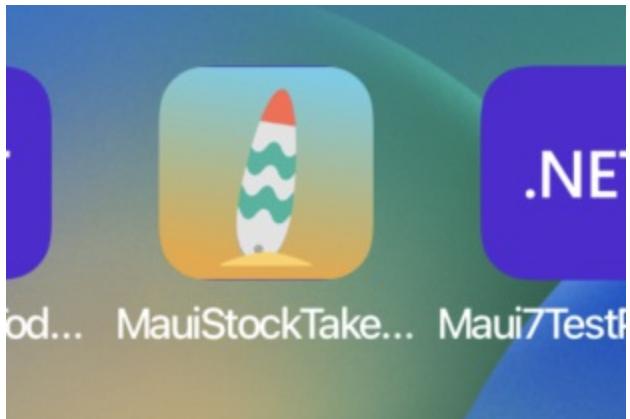
12.1.1 Replacing the default app icon

If you open `MauiStockTake.UI.csproj`, you'll see there is an icon defined in an `ItemGroup` marked with a comment that says `App Icon`. To define an app icon in .NET MAUI, you add the `MauiIcon` item and set `Include` to the file you want to use as your icon. You can use the `ForegroundFile` property to layer another image over the top.

Download the two icon files from the chapter 12 resources folder and copy them into the `Resources/AppIcon` folder in the `MauiStockTake.UI` project. We could update the `.csproj` file to refer to the two new filenames, but the most reliable way to use new files is to keep the default names. **Delete the `appicon.svg` and `appiconfg.svg` files**, then rename `icon-bg.svg` to `appicon.svg`, and rename `icon-fg.svg` to `appiconfg.svg`.

To ensure the new icons get deployed successfully, clean and rebuild your project, and delete the app from your device or simulator/emulator, then run it again. If you stop the app, you should see its icon has been updated, as in figure 12.3.

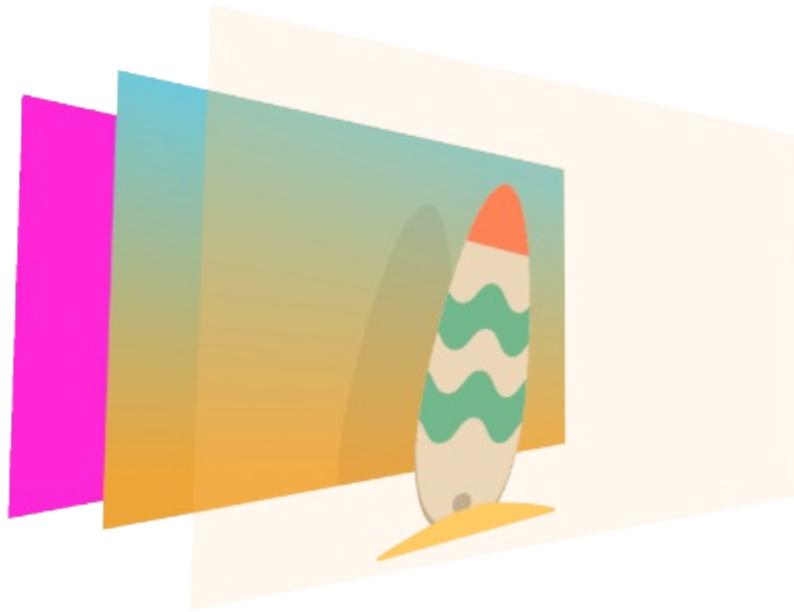
Figure 12.3 The `MauiStockTake` icon on iOS, made using the files provided rather than the default .NET icon, which you can see in another app next to it.



12.1.2 App icon composition

We've used two layers to compose our icon, but in fact you can build an icon out of four layers, as shown in figure 12.4.

Figure 12.4 An app icon in .NET MAUI can be composed of four layers: a background color, an icon, a foreground icon, and a tint color.



The four layers that can be used to compose an app icon in .NET MAUI are:

- A background color, specified by the `BackgroundColor` property
- An icon, specified with the `Include` property

- An icon foreground, specified with the `ForegroundFile` property
- A tint color, specified with the `TintColor` property

The background color ensures that your icon will be a solid bitmap even if there are transparencies in any of your layers, which is useful as iOS does not allow transparency in app icons. Both the `BackgroundColor` and `TintColor` properties can be specified using either hex colors or by using a statically defined .NET MAUI color (e.g. `Red`).

For MauiStockTake, we're only using the `Include` and `ForegroundFile` properties, so you can delete `Color` which is included by default.

12.1.3 App icon resizing

As your app icon is shown in a lot of places, it needs to be provided in a variety of sizes. This includes sizes for different device sizes and resolutions (from watches to TVs and everything in between), as well as store listings, search results, and notifications, as shown in figure 12.5.

Figure 12.5 An app icon is used in multiple places, including store listings, store details pages, device home screens, and notifications. It needs to be provided in sizes to accommodate all these scenarios and many different device sizes, including watches, phones, laptops and TVs.



Historically, generating the icon in all these different sizes has been a pain point of mobile development. Websites like <https://appiconmaker.co> can help, but even after you've generated copies of the icon in all the right sizes, you still have to import them all, which has also been a laborious process in the past.

.NET MAUI makes this much simpler by letting you add an icon, which is automatically resized and imported in all the different size specifications for each platform.

Resizerizer

Resizerizer is a package that was created by Jonathan Dick on the .NET MAUI team. Originally written for Xamarin.Forms, and later rewritten for .NET MAUI, resizerizer automatically scales an image for all required device resolutions. It's used to generate the app icon but is also used to automatically generate different scaled versions of your images.

If you've worked with Xamarin.Forms in the past, you'll know how difficult

it was to import and manage image assets. You had to provide different versions for each resolution, and for Android copy them to the correct device resolution folder, as well as updating the appropriate asset catalogue. A Visual Studio extension called MFractor simplified this process, but managing images without this paid add-on was a pain.

With .NET MAUI, this is all taken care for you. Simply add an image file to the `Resources/Images` folder (or otherwise specify the image's `BuildAction` as `MauiImage`), and .NET MAUI will use Resizer to generate all the required image assets for you.

Resizer will resize any image format, but it's best to use SVG where possible. This is because scaling of bitmap images can lead to pixelated or blurred results, but SVGs can be scaled up or down without any loss of fidelity. However, the resized images are all PNG files, which is why you must refer to images using the `.PNG` extension in your code, irrespective of the original format.

Having these icons resized automatically is an awesome feature, but if you want some extra control, you can specify a `BaseSize` property (in fact, if you use a bitmap image type, you *must* specify the `BaseSize` property for the image to be automatically resized). You can also disable resizing altogether by setting the `Resize` property to `false`.

You can find out more about these properties, and more about app icons in general in .NET MAUI including, for example, how to specify different icons per platform, by consulting the documentation:

<https://learn.microsoft.com/dotnet/maui/user-interface/images/app-icons>.

12.1.4 Replacing the default splash screen

Splash screens in .NET MAUI are composed in almost the same way as app icons, including autoscaling, with the exception that they only use one image layer. If you look in `MauiStockTake.UI.csproj`, you should be able to find the `MauiSplashScreen` item. We'll replace this with a design from Mildred's team too.

TIP

If your team doesn't include a designer or you don't have access to a design department, check out <https://www.svgrepo.com> and <https://www.reshot.com> (this is where I sourced MauiStockTake's icon and splash screen). They provide a massive range of attribution-free SVG images under the Creative Commons (or similar) license that you can use in your apps. For something unique, <https://www.fiverr.com> is a great resource that I have used for multiple personal projects.

Download the `splash.svg` file from the chapter 12 resources folder and copy it to the Resources/Splash folder in MauiStockTake.UI, overwriting the one that's already there. In `MauiStockTake.UI.csproj`, we're keeping the filename, but we need to update a couple of properties. **Change the `Color` property to #74A0B7 and change the `BaseSize` to 256, 256.**

Run the app now, and you should see the updated splash screen, as shown in figure 12.6.

Figure 12.6 The splash screen for MauiStockTake has been updated. The Mildred's Surf Shack logo is displayed in the foreground, and the background color has been set to the same colour used in the login screen (in light mode with the default theme).



Updating an app's icon and splash screen is a relatively simple process, but it instantly transforms your app from a development project to a professional product.

NOTE

If you're not seeing the expected results with the splash screen, try uninstalling the app from your device or emulator/simulator, and clean and rebuild the project before redeploying.

12.1.5 Application identifiers and version numbers

Each app needs a unique identifier to differentiate itself from other apps, both on devices and in the store ecosystems. The name alone is not sufficient, and more importantly, using an identifier distinct from the name allows the name to be changed.

On Android, iOS and macOS, a bundle ID is used. This is written in reverse-DNS format and typically identifies the app publisher and the app itself. For MauiStockTake, we'll set this to `com.mildredssurfshack.mauistocktake`.

IMPORTANT

these identifiers are case-sensitive.

NOTE

The publisher in this case should always be the business that the app is created for. In this case we are using Mildred's Surf Shack as the publisher, not Beach Bytes. By the same token, the developer account (covered later in this chapter) used to publish the app should be owned by Mildred's Surf Shack, and not Beach Bytes.

On Windows, the application is identified with a GUID, which is generated automatically by the template when you create a new .NET MAUI project, so we don't need to update this. But let's fix up the bundle ID for the other platforms, and while we're at it, we can fix up the display name of the app too.

Open `MauiStockTake.UI.csproj`, and update the `ApplicationTitle` and `ApplicationId` items, as shown in listing 12.1

Listing 12.1 the app identifiers

```
<!-- Display name -->
<ApplicationTitle>MauiStockTake</ApplicationTitle> #A

<!-- App Identifier -->
<ApplicationId>com.mildredssurfshack.mauistocktake</ApplicationId>
<ApplicationIdGuid>F31A0539-B7DB-4874-94A9-489AA23BDF47</ApplicationIdGuid> #C
```

If you run the app now, you'll note that you have two versions of the app on your device or emulator/simulator. This is because, now that you've changed the identifier, the OS considers it to be a different app. You can safely uninstall the old one, or if you can't easily determine which is which, just delete both until the next time you deploy.

The application ID lets the store ecosystems and devices identify your app, but they also need to distinguish different versions of your app too. This allows you to provide updates when you have new features and bug-fixes. Just like app identifiers, version identifiers have a reference version, used to identify the version of the app (analogous to the `ApplicationId`) and a display version, used to indicate to the user which version they are using (analogous to the `ApplicationTitle`).

These are handled differently on the different platforms, but .NET MAUI provides a unified way of setting them, also in the `.csproj` file. In `MauiStockTake.UI.csproj`, find the two properties next to the `Versions` comment, as shown in listing 12.x2.

Listing 12.2 Application versions

```
<!-- Versions -->
<ApplicationDisplayVersion>1.0</ApplicationDisplayVersion>
<ApplicationVersion>1</ApplicationVersion>
```

`ApplicationDisplayVersion` is the version displayed to users, and `ApplicationVersion` is the property used by operating systems and stores to identify new builds. It's useful to be able to set these independently. For example, you may go through several builds that you need to deploy to test users, and you'll need to increment the version number so that they get the new build, but you may not want to update the display version until you have a new public version that you're shipping.

You don't need to update either of these now. With a proper application identifier, custom icon, and custom splash screen, your app has the final bits of polish it needs to start distributing it to users.

12.2 Deploying Apps with GitHub Actions

There are several ways to build and deploy your apps to the various stores. We can use the `dotnet publish` command in the .NET CLI, or we can right-click-publish in Visual Studio (on macOS or Windows). I've got a comprehensive guide to manually deploying your apps on my blog, which you can see here: <https://goforgoldman.com/posts/maui-app-deploy/>.

If you haven't published mobile or desktop apps via the stores before, I recommend working through the blog posts and understanding the process of manually deploying apps. In all cases you need to upload at least one build manually, and for Windows you have to actually publish (i.e., release to the public) your app before you can use automation to deploy it.

However, as with any enterprise application, the best approach is to use automated build tools and deploy your app with a CI/CD pipeline. We're going to use GitHub Actions for this, as it's one of the most popular, and likely what you will use in your professional work. In any case, while there may be some slight variations with other tools, the workflows we create in this chapter should be straightforward to translate to other platforms.

If you've never used GitHub Actions before you'll still be able to follow this chapter and should be able to grasp what we cover, but for a better understanding of some of the concepts we don't dive into, I recommend the 20-minute GitHub Actions course on Microsoft Learn:

<https://learn.microsoft.com/training/modules/introduction-to-github-actions/>.

12.2.1 Setting up the workflow

Part of the popularity of GitHub Actions workflows is due to the simplicity of setting them up and maintaining them. You simply add a YAML file to your code repository in a certain location, and GitHub will automatically execute the instructions in the file. YAML (short for "YAML Ain't Markup Language") is a human-readable data serialization format that is commonly used for CI/CD pipelines. If you're not familiar with it, don't worry; it's designed for simplicity, and you'll have no problem following the example in this chapter.

To add a GitHub Actions workflow, in the root of your repository, create a

folder called `.github`, and in here another folder called `workflows`. We place our GitHub Actions workflows in this folder (defined in YAML) and GitHub will automatically execute them.

Create the `workflows` folder now, and in here create a file called `build-and-deploy.yaml`.

Working with YAML files

The easiest way to work with YAML files is with a text editor (especially a developer-focused one like Visual Studio Code). This is part of the simplicity of YAML: it's just text files that are very easy to structure and read.

Visual Studio 2022 introduced a feature that lets you work on GitHub Actions within your solution. Historically this wasn't possible, as the period prefix on the `.github` folder marks it as hidden, but this new update lists GitHub Actions in Solution Explorer.

At the time of writing, this is a preview feature, and **I do not recommend installing preview versions of Visual Studio** if you are working with .NET MAUI; at least, not on your main development machine. Preview updates to .NET MAUI are included with preview versions of Visual Studio and can introduce breaking changes, even to stable versions, that can be very difficult to recover from.

VS Code is my preferred way of working with YAML (in fact it's my preferred way of working with any code that isn't .NET) and is the tool I recommend for working through this chapter.

We're going to add three high-level sections to our GitHub Actions workflow file:

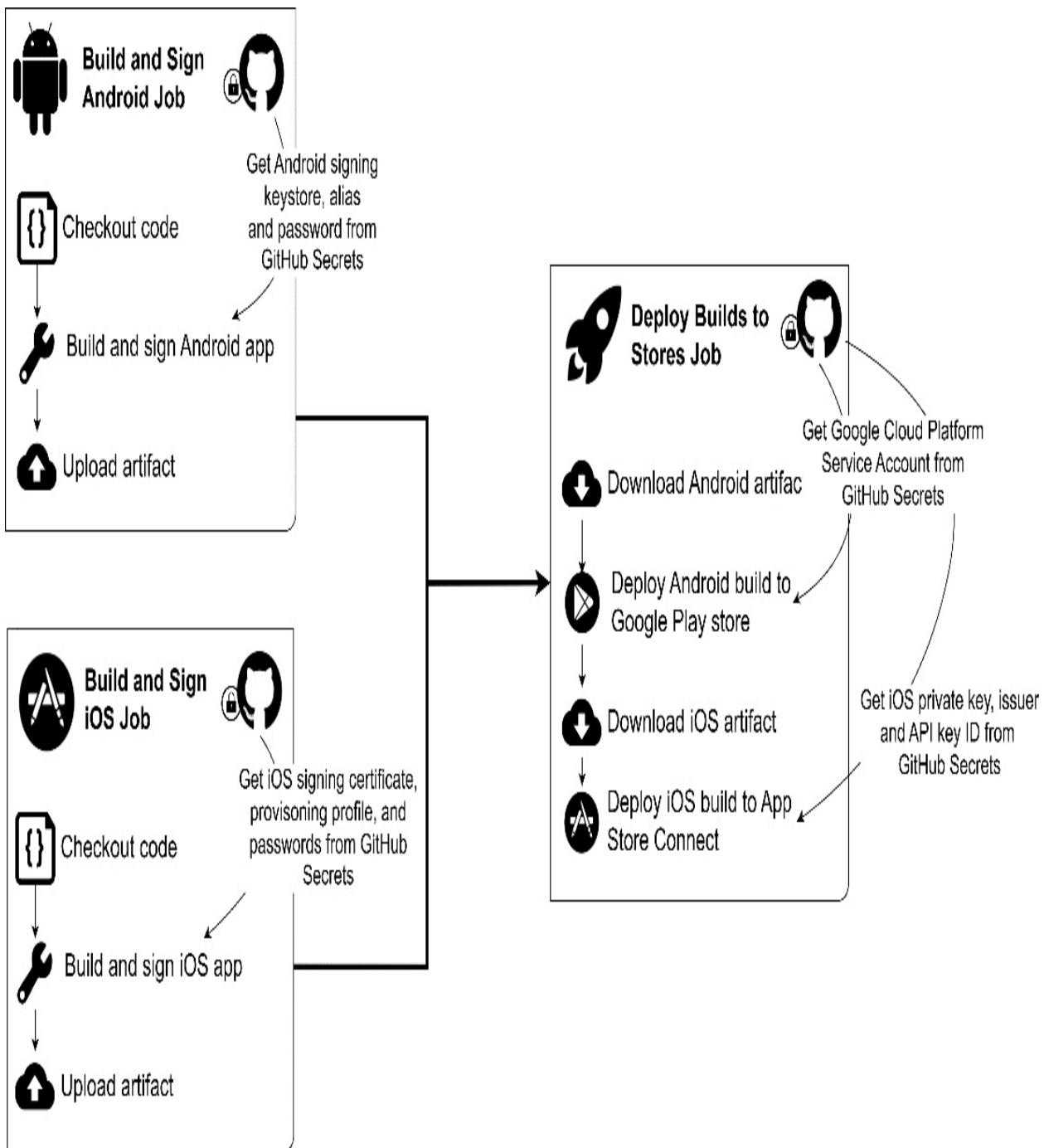
- **Name:** This is the name of the workflow. In this case we can call it `Build and Deploy MauiStockTake`.
- **On:** This section defines the triggers that will cause the workflow to be run. You can specify which branches and which activities (e.g., pull request, or code push) will trigger the workflow, as well as the `workflow_dispatch` trigger which allows the workflow to be run

manually.

- **Jobs:** This section defines the actual work that the workflow will do. Ours will consist of three jobs: one to build the Android app, one to build the iOS app, and one to deploy the builds to the stores.

We need to use some sensitive data in our workflow, such as passwords and signing keys, and we can take advantage of GitHub Actions secrets for this. Using secrets, GitHub securely stores and encrypts data to make it available in workflows. The workflow engine is also smart enough to recognize when a secret is in use and redacts it in any console or log output. Figure 12.7 shows an outline of the three jobs in our workflow.

Figure 12.7 The GitHub Actions workflow is made up of three jobs. One builds and signs the Android app, another builds and signs the iOS app, and a third one deploys the two signed builds to their respective stores. The deploy job is dependent on the two build jobs so they must complete successfully for it to run. All jobs depend on GitHub Actions secrets for signing keys, passwords, and other sensitive data.



To set up the workflow, we need to define the name and the triggers, then we can add the jobs. Open the `build-and-deploy.yaml` in your text editor, and to define the workflow's name, on the first line add:

```
name: Build and Deploy MauiStockTake
```

YAML files use a combination of colons, whitespace, and indentation to

indicate the structure of the data. Colons are used to separate keys and values, while whitespace and indentation are used to indicate nested elements and lists.

Next up let's add the triggers. The triggers in a GitHub Actions workflow are identified by the `on` key, so we'll add `on:` and then indent the following lines to indicate that they're part of this section. We'll add two sections to this:

- `push`: This trigger indicates that the workflow should run when code is pushed to the nominated branches. It will also be triggered when a pull request is completed to the nominated branches (i.e., when the code is merged).
- `workflow_dispatch`: This trigger indicates that the workflow can be triggered manually from the GitHub website.

Tests

In a real-world app, you would also typically include unit tests (or other pertinent tests). You could add another workflow that includes the `pull_request` trigger, that would run your tests. You could then make passing tests a requirement for merging pull requests.

Alternatively, you can also include all the triggers in the one workflow and define within each job what conditions must be met for them to run. For example, your tests and builds might always run, but your deploy would only run when code is merged into your main branch.

For the `push` trigger, we'll nominate the main branch, so that the workflow won't run every time we push a commit to our feature branches. We don't need this for the workflow dispatch trigger; it will always be run manually against the main branch.

With the triggers added, the `build-and-deploy.yaml` file will now look like listing 12.3.

Listing 12.3 The build-and-deploy.yaml file

```
name: Build and Deploy MauiStockTake      #A
```

```

on:                      #B
  push:                  #C
    branches:             #D
      - main               #E
  workflow_dispatch:     #F

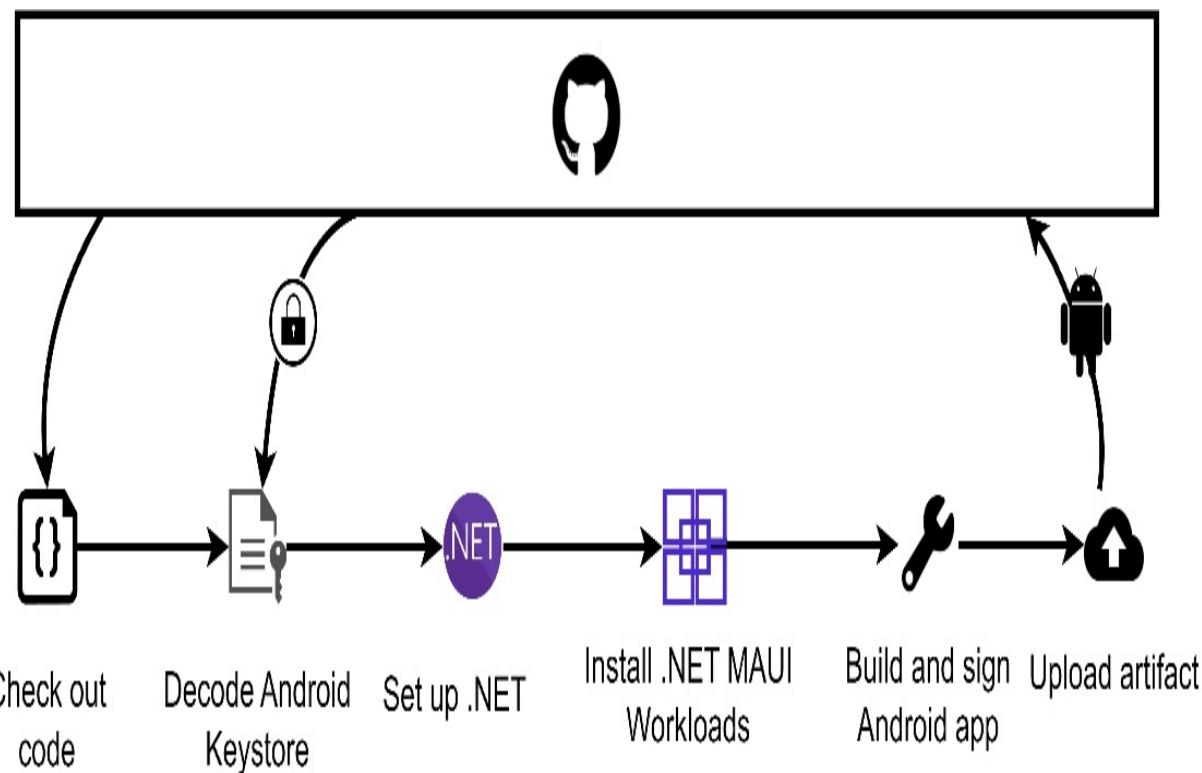
```

Our workflow is now named and has its triggers defined, so we're now ready to start adding jobs.

12.2.2 Build and Sign the Android Job

The first job we will add to our workflow is to build and sign the Android app. This job is summarized in figure 12.7, but it will actually consist of six steps, as shown in figure 12.8 below.

Figure 12.8 The first step in this job checks out the code from the repo. The next will retrieve the Android signing keystore from GitHub Actions secrets and convert it from a base64 string. The next two steps set up dependencies (.NET and the .NET MAUI workloads). After that the next step creates a signed build, and the final step uploads the signed build to GitHub, ready for the deploy job to pick it up later.



To sign Android apps we need a keystore. If you've followed the steps in the blog post linked above, or otherwise created a keystore using Visual Studio, on Windows, you can find this keystore file in

%LocalAppData%\Xamarin\Mono for Android, and on macOS you can find it in ~/.local/share/Xamarin/Mono for Android/. You can also create the keystore from the command line using the keytool command:

```
keytool -genkeypair -v -keystore mauistocktake.keystore -alias ma
```

This will create a keystore using the RSA algorithm with a 2048-bit key size. You will be prompted to provide some details and a password, end will end up with a file called `mauistocktake.keystore`.

WARNING

From Google's point of view, this key store *is* your Android app. If you lose the key store or the password, you will not be able to upload new versions of your app to Google Play. I have heard of Google support helping people around this, but it is not something you should depend on.

The keystore, password and alias are confidential and must be secured, which means we can't check the keystore into the repo, and we can't put the password and alias in plain text in the workflow. Fortunately, we can use GitHub Actions secrets to protect these for us; however, secrets can only be strings, and we need to store files as well as strings. To overcome this, we can encode the files as base64 strings and upload them that way.

In the Android job, the second step is decoding the keystore, from the base64 encoded string to a file. We're going to run this job on a Windows agent and use PowerShell to decode the string to a file, so to ensure consistency we can use PowerShell to encode the file.

NOTE

if you are using macOS, you can follow the instructions for base64 encoding files in section 12.2.3. Or you can install PowerShell Core onto your Mac.

Listing 12.4 shows the PowerShell commands to execute to convert your

keystore file to a base64 encoded string.

Listing 12.4 Converting files to base64 in PowerShell

```
$keystore = Get-Content mauistocktake.keystore -Encoding Byte  
$base64 = [System.Convert]::ToBase64String($keystore)  
$base64 | Out-File keystore_file_b64.txt
```

Note that this is not a script, it's three individual commands to run, and assumes that your keystore file is called `mauistocktake.keystore`. Once you have run it, you'll have a text file called `keystore_file_b64.txt`. The content of this file is the keystore itself encoded as a base64 string, which you will be able to upload to GitHub.

Now that all our secrets are prepared for this job, let's upload them to GitHub. Go to your code repository, and go the **Settings** tab. Expand **Secrets and variables** on the left, then click **Actions**. Upload the Android secrets, using the table below as a guide, following the GitHub Actions convention of uppercase words separated by underscores.

Table 12.1 The secret names and values to store in GitHub Actions secrets for the Android job

Secret name	Value
ANDROID_KEYPASSWORD	This is the password for the signing key within the keystore
ANDROID_KEYSTORE	This is the keystore file encoded as a base64 string
ANDROID_KEYSTOREALIAS	This is the alias of the keystore
ANDROID_KEYSTOREPASSWORD	This is the password for the keystore



IMPORTANT

When uploading files to secrets encoded as base64 strings, remove any extra line breaks from the end. The secret should end on the last character of the base64 string.

Now that we've got the secrets set up for the steps in the Android job, let's start adding the steps. The first thing to do is to define the jobs section within the workflow, and then the job by giving it a key. We can then give the job a title and define what kind of agent will be used to run it. In this case, we'll use a Windows runner, hosted by GitHub, as the agent. Listing 12.5 shows the code to add to the `build-and-deploy.yaml` file. Add this after the `workflow_dispatch` line, with an extra line break in between.

Listing 12.5 The Android job definition

```
jobs:  
  build-android:  
    name: Build Android  
    runs-on: windows-latest
```

As we've seen before, indentation is used to indicate structure and hierarchy in a YAML file, so here we can see that we have a `jobs` section, and in here we have a job defined with the `build-android` key. This job has a `name` property (this is friendlier than the key and will be shown in the workflow results) and a `runs-on`, which we have defined as `windows-latest`, meaning it will run on a GitHub hosted Windows agent running the latest version.

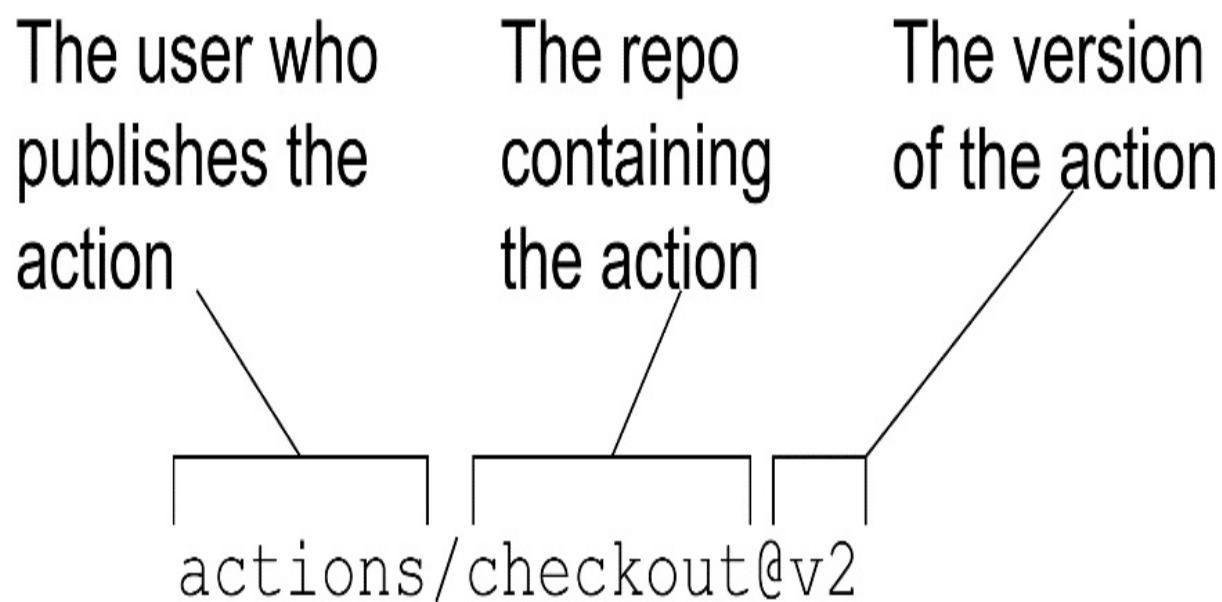
The 'actions' in GitHub Actions are the steps that we will add in the next section of the workflow file. Actions can be run in one of two ways. The first is with the `run` command, which lets you execute arbitrary shell commands. You can of course use these to call your own scripts or executables.

NOTE

The default shell for Windows runners is PowerShell, for Linux runners is bash, and for MacOS runners is zsh; but you can use a different shell if you need to.

You can also run predefined actions published by users on GitHub. These are run with the uses key instead of run, and referred to in the format [user]/[repo]@[version], as shown in figure 12.9.

Figure 12.9 GitHub Actions are referred to in workflows by reference to the user, then the repo where the action is hosted, then the version of the action.



GitHub hosts several common actions, from a user account called actions. You can see them all by browsing the repositories from the profile <https://github.com/actions>, and many other users make actions available, some of which we'll use in our workflow. You can explore the publicly available actions for use in GitHub workflows at the GitHub marketplace: <https://github.com/marketplace?type=actions>.

The first step in the workflow is to check out the code, and GitHub provides an action for this. In the second step, we'll retrieve the base64 encoded keystore from GitHub Actions secrets and convert it to a file, and we'll do this using shell commands. Secrets can be accessed using a placeholder syntax, in the format `${{ secrets.[YOUR_SECRET_NAME] }}`.

We'll use a GitHub Action to set up .NET in the third step, and in the fourth step we'll use shell commands to install the .NET MAUI workloads we need to build our app. Listing 12.6 shows the steps section of the Android job with the first four steps.

Listing 12.6 The first four steps of the Android job

```
steps:                                #A
  - name: Checkout                      #B
    uses: actions/checkout@v2           #C

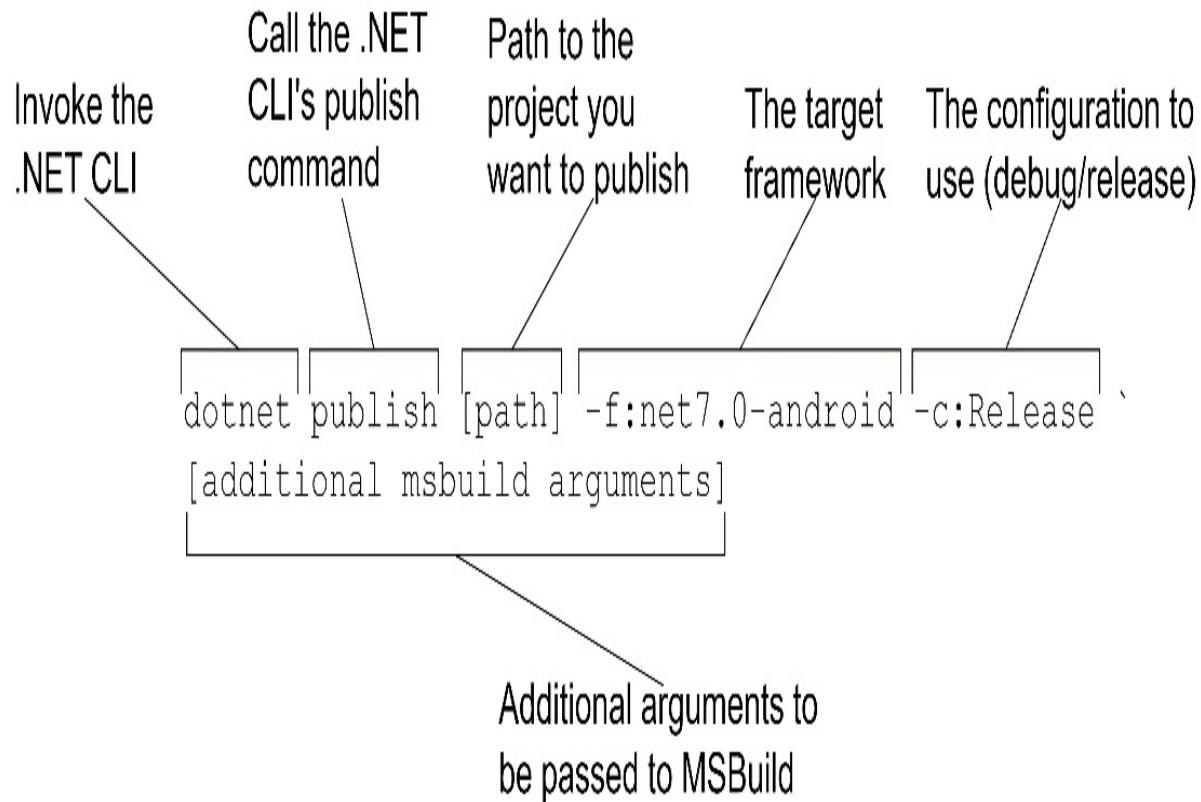
  - name: Decode keystore file
    run: |
      $keystorebase64 = "${{ secrets.ANDROID_KEYSTORE }}"
      $keystore = [System.Convert]::FromBase64String($keysto
      Set-Content "${{ github.workspace }}/mauistocktake.keys

  - name: Setup .NET
    uses: actions/setup-dotnet@v1
    with:
      dotnet-version: 7.0.x             #F

  - name: Install MAUI workload
    run: dotnet workload install maui      #G
```

This puts all the pieces in place we need to build and sign the Android app. For this step, we're going to call the .NET CLI from a shell command. We'll simply call `dotnet publish` and use the Release configuration, specifying the Android target framework. We've seen most of the pieces of this in chapter 2; in this case we're substituting `publish` for `build`. The anatomy of the `dotnet publish` command is shown in figure 12.10.

Figure 12.10 The `dotnet publish` command is similar to the `dotnet build` command we saw in chapter 2. You provide the path to the project to be published, the target framework and configuration, and any additional MSBuild arguments.



Under the hood, the `dotnet publish` command uses `MSBuild`, and `MSBuild` accepts the arguments we need to sign the Android build. Arguments are passed to `MSBuild` by using the `p:/[argument]` syntax, and we will need to pass the following arguments:

- **AndroidKeyStore**: This is a Boolean flag to indicate that we will use a keystore.
- **AndroidSigningKeyStore**: This is the path to the keystore itself, so it will be the keystore file in the `github.workspace` directory.
- **AndroidSigningKeyStorePass**: This is the password to the keystore file.
- **AndroidSigningKeyPass**: This is the password for the signing key inside the keystore.
- **AndroidSigningKeyAlias**: This is the alias that identifies the signing key within the keystore.

The `dotnet publish` command doesn't support these arguments directly, but we can pass them through to `MSBuild` to get the desired effect. With this information, we're ready to add the build and sign step to our workflow. You

can see this step in listing 12.7.

Listing 12.7 The build and sign Android step

```
- name: Build  
  run: dotnet publish src/Presentation/MauStockTake.UI/Mau
```

The final step for the Android job is to upload the signed AAB file to GitHub, ready to be pulled down by the deploy job. GitHub provides an action called `upload-artifact` that serves exactly this purpose, so we'll use the `uses` key instead of `run` to invoke this action. Some actions take additional parameters, and we can pass these using the `with` key. The parameters we will use with the `upload-artifact` action are `name` and `path`. The `name` property will specify the name to use to retrieve the artifact in a later job, and the `path` specifies the path to the file to be uploaded.

This gives us the final step in the Android job, which you can see in listing 12.8.

Listing 12.8 The upload artifact step in the Android job

```
- name: Upload Android artifact  
  uses: actions/upload-artifact@v3.1.0  
  with:  
    name: mauistocktake-android-build  
    path: src/Presentation/MauStockTake.UI/bin/Release/net
```

NOTE

I'm using an asterisk (*) in the file path rather than specifying the full AAB filename. This is just for convenience, but also lets me easily adapt this to other workflows for other apps. You could also use a glob pattern like `**/*.aab` which would make it work with any repo, but for the avoidance of doubt it's better to be at least a little specific.

This completes the Android job. You could run the workflow now and should see a green tick for the Android job. If it fails, check the log for any errors, and address them as necessary.

12.2.3 Build and Sign iOS Job

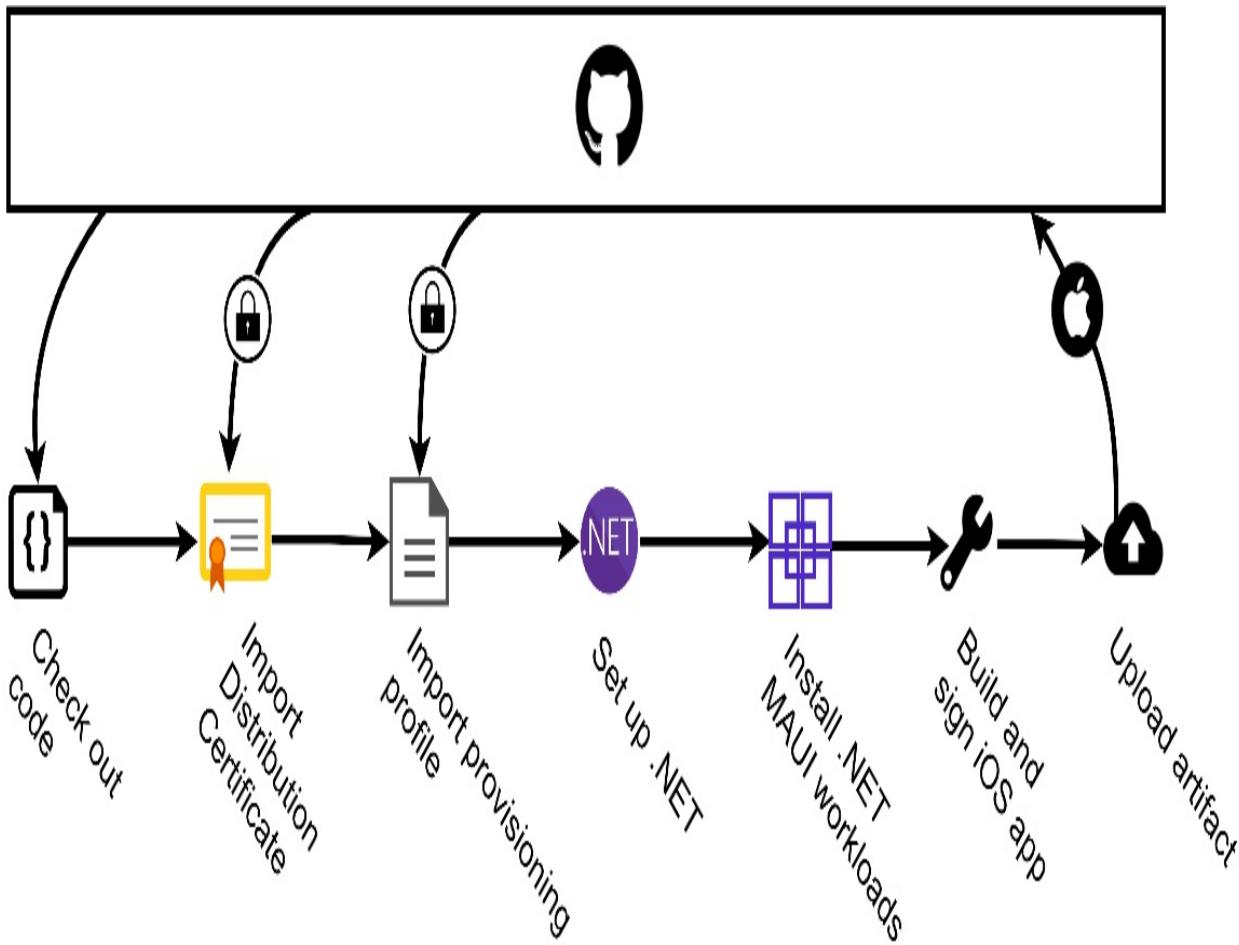
The job to build and sign the iOS app is similar to the Android job, in that we check out the code, decode a file (in this case the distribution certificate as opposed to the keystore), build and sign the app, and upload the build to be picked up later by the deploy job.

NOTE

If you're not already familiar with manually building, signing and deploying iOS apps, I recommend reading up on it on my website here:
<https://goforgoldman.com/posts/maui-app-deploy-2/>.

The iOS job has some subtle differences though. In addition to the signing certificate, we also need to decode and install the provisioning profile. You can see a summary of the steps for the iOS job in figure 12.11

Figure 12.11 The iOS build and sign job is similar to the Android job, but decodes (and installs) a distribution certificate instead of a keystore. It will also need to decode and import the provisioning profile.



We will also make a choice about where to run the iOS job; as it requires a macOS agent to run, a self-hosted runner is a better choice than a GitHub hosted runner. You can see the instructions for setting up a self-hosted runner here: <https://docs.github.com/en/actions/hosting-your-own-runners/adding-self-hosted-runners>. The workflow will also run on a GitHub hosted runner, with a couple of small changes. I'll include these too, but they will be commented out. Uncomment these if you're using a GitHub hosted runner.

Self-hosted vs GitHub hosted runners

GitHub Actions workflows can run on Linux, Windows or macOS agents. These can either be hosted by GitHub, or you can host them yourself and link them to your repository or organization.

For workflows that require macOS, I recommend a self-hosted runner rather

than a GitHub runner. This is due to the limits imposed by GitHub: for most accounts you get 2,000 minutes per month (the Enterprise plan gets you 50,000), but this is the baseline for Linux runners. For Windows runners a 2x penalty is imposed, and for macOS it is 10x. This means that a job that takes 50 minutes will eat a quarter of your monthly quota in one hit.

Workflows can fail, and sometimes have unpredictable results, and consequently it's easy to burn through your entire quota without getting a single successful build. By using a self-hosted runner, you avoid these problems as you don't have any time limits in this sense (a single workflow run cannot exceed 35 days, but if your workflows run this long you've got other problems).

It may be that you specifically want to use GitHub as a build agent because you don't have a Mac. Using a GitHub hosted runner can be a potential workaround, however the problems mentioned above will be exacerbated if you haven't had the opportunity to test your build on a Mac before pushing to GitHub. While it's theoretically possible to develop, build and deploy a .NET MAUI app to iOS or macOS without a Mac, your developer experience will be much smoother if either you or someone on your team has one.

An entry level Mac Mini is a good investment; they are relatively inexpensive, will make your life as a developer much easier if you're targeting macOS or iOS, and they can double as a self-hosted runner for GitHub Actions.

Let's add the iOS job to the workflow. To start it will be almost identical to the Android job: it will have a key to identify it in the file, a name and a runs-on property. We'll set name to Build_iOS and runs-on to self-hosted. You can change this if you've named your runner something different, or macos-latest if you want to run it on GitHub. If you do run it on GitHub, add a timeout property; this is essential as if something goes wrong with your workflow, it could get stuck running and will blow out your available GitHub Actions minutes.

After that, just like with the Android job, we'll check out the code from the repo. Listing 12.9 shows the start of the build iOS job.

Listing 12.9 The setup for the build iOS job

```
build-ios-mac:  
  
  name: Build iOS  
  runs-on: self-hosted  
  # timeout-minutes: 30          #A  
  
  steps:  
    - name: Checkout  
      uses: actions/checkout@v2
```

Before we can proceed any further, we need to set up the secrets that the build iOS job will need. Let's start with the certificate. As with the keystore in Android, as it is a file, we will need to encode it to a base64 string to store it in GitHub Actions secrets. But first, we need to export it from the Keychain app. Open the **Keychain Access** app and find your distribution certificate. You can do this easily by searching for `Apple Distribution`.

Once you've found the certificate, select it and the corresponding private key (it should be listed directly below it), **right-click them and select Export 2 items...** Ensure that the .p12 format is selected (you can also change the name if you want, but it doesn't matter as we won't be using this file in the workflow), and click **Save**. You'll be prompted to enter and confirm a password; keep these safe as we will need to upload them to GitHub. You will also need the name of the certificate for use in the workflow; double-click on the certificate to see its details, then highlight the whole name (starting with `Apple Distribution...` and ending with the team name (the random string) and closing bracket). Copy this and paste it somewhere you can refer to it later.

The .p12 file contains both the certificate and the private key, and as such should be considered sensitive and handled accordingly (once I've uploaded the base64 encoded version I delete it; I can always export it again). We'll use some zsh commands to encode the file, to ensure consistency with the environment that will be used to decode them.

TIP

The Terminal app that ships with macOS is ok, but you can get a better

experience with an enhanced terminal. iTerm is a popular terminal app, but I've become a big fan of Warp. Find out more here: <https://warp.dev>.

The zsh shell includes a base64 command that we can use to encode the certificate. It takes an input and an output parameter, denoted by -i and -o respectively, that identify the file to be encoded and the resulting output file. Encode the certificate and private key using the following command:

```
base64 -i certificate.p12 -o certificate-base64.txt
```

In addition to the certificate, we need the provisioning profile. This is a little more straightforward to obtain as you can simply download it from the Apple Developer website (go to <https://developer.apple.com>, click on Account to log in, then go to Profiles). Download the correct distribution profile, and use the base64 command to convert it to a base64 encoded string:

```
base64 -i MauiStockTake_iOS_Distribution.mobileprovision -o profi
```

Now that we've got all our secrets, let's add them to GitHub. Table 12.2 below shows you the secret names to use and the values to copy in for each.

Table 12.2 The secret names and values to store in GitHub Actions secrets for the iOS job

Secret name	Value
APPLE_CERT	This is the certificate and private key. Copy the content from the certificate-base64.txt file.
APPLE_CERT_PASSWORD	This is the password for the .p12 file that you set when you exported it.
APPLE_CERT_NAME	This is the name of the certificate that you copied from Keychain Access.

APPLE_PROFILE	This is the provisioning profile. Copy the content from the profile-base64.txt file.
APPLE_PROFILE_NAME	This is the name of the distribution profile, as it appears in the Apple Developer website.

Now that the secrets are in GitHub, let's add the remaining steps. The first two after checkout will be to decode and install the certificate, and to decode and install the provisioning profile. We can use the same base64 command that we used in our terminal, but with the --decode flag to indicate that we want to decode a base64 string, rather than encode some data to one.

For both these steps, we'll use the run key and supply shell commands. For both steps, on the first line we can echo the content of the relevant secret and pipe it to the base64 command, then output it to a file. For the certificate, we can use the security command to import the certificate, and for the profile we will need to create the appropriate directory, then copy the profile to it.

Listing 12.10 shows the two steps to import the certificate and provisioning profile.

Listing 12.10 The steps to import the ceritficate and provisioning profile

```
- name: Import Distribution Certificate
  run: |
    echo ${{ secrets.APPLE_CERT }} | base64 --decode > Dist
    security import DistributionCertificate.p12 -k ~/Library
      Keys/Distribution.p12

- name: Import Provisioning Profile
  run: |
    echo ${{ secrets.APPLE_PROFILE }} | base64 --decode > M
    mkdir -p ~/Library/MobileDevice/Provisioning\ Profiles
    cp MauiStockTake_iOS_Distribution.mobileprovision ~/Lib
```

The remaining four steps are the same as the last four steps in the Android workflow: we setup .NET, install the .NET MAUI workload, and build and sign the app, then upload the artifact. The differences are in the build and sign step; the target framework will be iOS instead of Android, and the MSBuild arguments that we pass will be different.

Before we move on to these steps, there's one extra step that's required for a GitHub hosted runner, and that is to specify the XCode version. If you're using a self-hosted runner, you just need to ensure the correct version of XCode is installed.

At time of writing the version you need to sign iOS apps built with .NET MAUI is 14.1, but you may need to update this in the future. Listing 12.11 shows the step to select the XCode version. It is commented so you can copy it into your workflow regardless of whether you're self-hosting or running on GitHub. If you're running on GitHub, uncomment this step.

Listing 12.11 also includes the steps to set up .NET and install the .NET MAUI workloads (they are identical to those in the Android workflow). Include these irrespective of which runner you are using.

Listing 12.11 The select XCode version step

```
# - name: Set XCode Version
#   if: runner.os == 'macOS'
#   shell: bash
#   run: |
#     sudo xcode-select -s "/Applications/Xcode_14.1.app"
#     echo "MD_APPLE_SDK_ROOT=/Applications/Xcode_14.1.app"

- name: Setup .NET
  uses: actions/setup-dotnet@v1
  with:
    dotnet-version: 7.0.x

- name: Install MAUI workload
  run: dotnet workload install maui
```

As mentioned above, the publish step is the same as in the Android workflow, just with the target framework and MSBuild arguments changed. The MSBuild arguments we need for iOS signing are:

- **ArchiveOnBuild**: This is a flag that tells MSBuild to create an ipa file (an iOS archive) on build, so will be set to true.
- **RuntimeIdentifier**: While we are providing the target framework, we also need to tell MSBuild the runtime identifier so it can be matched against the profile. Specify this as `ios-arm64`.
- **CodesignKey**: This is the name of the certificate that we copied from Keychain Access (not the name of the file). We've already added this as a secret, so we'll refer to that secret here with the placeholder syntax.
- **CodesignProvision**: This is the name of the provisioning profile. We have also added this as a GitHub secret so can refer to it here with the placeholder syntax.

Just like with the Android build, these arguments are passed through to MSBuild with the `/p:` flag. Listing 12.12 shows the remainder of the steps for the iOS job. All of these are the same as the Android job, except for the minor changes to the `Build` step, and the file path and glob to upload the artifact.

Listing 12.12 The remaining steps for the iOS job

```
- name: Setup .NET
  uses: actions/setup-dotnet@v1
  with:
    dotnet-version: 7.0.x

- name: Install MAUI workload
  run: dotnet workload install maui

- name: Build
  run: dotnet publish src/Presentation/MauiStockTake.UI/Mau

- name: Upload iOS artifact
  uses: actions/upload-artifact@v3.1.0
  with:
    name: mauistocktake-ios-build
    path: src/Presentation/MauiStockTake.UI/bin/Release/net
```

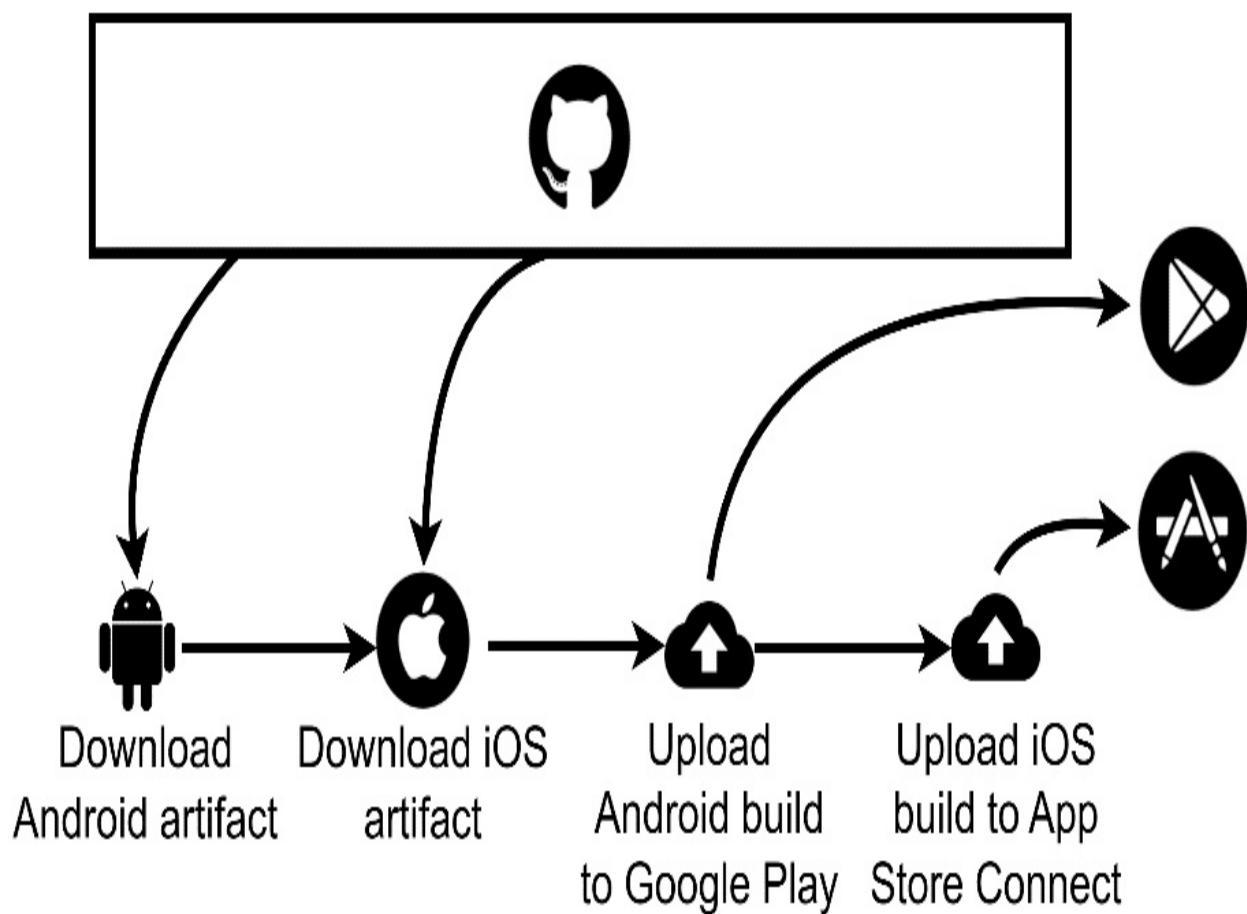
We've now got a workflow that builds and signs our Android app and our iOS app. All that's left is to deploy it to the stores.

12.2.4 Deploy to Stores Job

The final job in the workflow will deploy the signed builds to the Google Play store and to App Store Connect. The Google Play store and App Store Connect have APIs that we can use to upload these artifacts programmatically, and there are community provided actions that we can use in the job that call them.

Figure 12.12 shows a summary of the main steps in the deploy job.

Figure 12.12 The deploy job is relatively simple and has four main steps: download the Android artifact, download the iOS artifact, upload the Android artifact to Google Play, and upload the iOS artifact to App Store Connect.



NOTE

There's one additional step not shown in figure 12.12, but this is just to put one of the filenames into an environment variable for later reference.

To use these APIs, we need to generate credentials that can be used to upload our builds. We'll step through these first, then come back to the workflow.

Generating Google Credentials

To use the Google API, we need to create a service account in the Google Cloud Platform (GCP). The first step is to link a GCP project to the Google Play Console. I cover these steps in the article I linked to above, specifically at the start of the section on deploying from Visual Studio (<https://goforgoldman.com/posts/maui-app-deploy/#deploying-from-visual-studio.>)

It's a good idea to complete the end-to-end process detailed in the linked article, to ensure that you can successfully deploy a build before you burn through any of your GitHub Actions minutes.

Once that's done, on the **API Access** page in the Google Play console, scroll down to the **Credentials** section. Next to the **Service accounts** sub-heading is a link that says **Learn how to create service accounts**. This will open a dialog that details the steps you need to follow, along with a link to the appropriate area of GCP. What's not listed is the **role** you need to assign to the service account; scroll through the list of roles until you get to the Service Accounts category, then select the **Service Account User** role and complete the steps as described.

WARNING

I've seen some recommendations to use the Owner role, as this will guarantee that the service account has the necessary permissions. But this violates the principle of least privilege and is a security risk.

Once you've created the service account you should see it listed in a table. Click on the service account link to see its details, then click on the **KEYS** tab. Expand the **ADD KEY** dropdown and select **Create new key**. A dialog will appear offering you two options. Accept the recommendation for **JSON** and click **CREATE**. The GCP will now generate a JSON file and automatically trigger a download.

Upload the content of the JSON file as a GitHub Secret. As it's JSON, it's just a text string and doesn't need to be base64 encoded. Add it as a secret named `GCP_SERVICE_ACCOUNT`, and don't forget to remove any line breaks from the end.

Creating Apple Credentials

We'll need to create App Store Connect API credentials to upload our build. Log in to App Store Connect and go to the **Users and Access** section. Go to the **Keys** tab and click on the plus (+) button. Enter a suitable name, such as `MauiStockTake Upload Key`, and from the **Access** dropdown select **App Manager**.

Click the **Generate** button, then click on the **Download API Key** link next to your newly created key in the table. You'll see a warning saying that an API key can only be downloaded once (if you lose it you'll have to generate another one); click the **Download** button. App Store Connect will download a file with a `.p8` extension, but it's just a text file, so you can open it in any text editor.

You'll also need the **Issuer ID** from this screen; conveniently, it's got a **copy** button next to it that you can click to copy it to your clipboard. You also need the **KEY ID**, when you want to copy this, simply highlight it and copy it as you would any other text you've selected.

We've now got all the details we need for the Apple upload step. Table 12.3 below shows the secret names to use in GitHub Actions secrets and what values to assign to them.

Table 12.3 The secret names and values to store in GitHub Actions secrets for the Apple upload step

Secret name	Value
	This is the issuer ID you copied from

APPSTORE_ISSUER_ID	App Store Connect.
APPSTORE_API_KEY_ID	This is the key ID you copied from App Store Connect.
APPSTORE_API_PRIVATE_KEY	This is the content of the .p8 file that you downloaded.

Now that we've got all of the Google and Apple secrets, we're ready to complete the workflow.

Completing the Workflow

As described in figure 12.12, the deploy job will consist of four main steps. One of these steps, the step to upload to App Store Connect, depends on macOS, so this job we'll also run this job on our self-hosted runner (or a macOS runner if you need to run it on GitHub).

The job will start the same way as the others: with a key, a name property and a runs-on property. We'll add one other key, which is needs, under which we can list the jobs that must complete successfully before this one. This serves two purposes, the first of which is that it allows some jobs to be run in parallel, but more importantly, it saves us from running this job unnecessarily if either of the build and sign jobs have failed.

Listing 12.13 shows the start of the deploy job.

Listing 12.13 The start of the deploy job

```
deploy-all:
  name: Deploy builds to stores
  runs-on: self-hosted
```

```
needs:          #A
  - build-android-windows
  - build-ios-mac
```

Next we'll add the steps, and the first two steps will download the Android and iOS artifacts. For this we will use a GitHub-provided action, that acts as the inverse of the upload artifact action, using the name that we used in those upload steps to identify the artifacts. Listing 12.14 shows the start of the steps portion of the job.

Listing 12.14 The first two steps of the deploy job

```
steps:
  - name: Download Android artifact
    uses: actions/download-artifact@v2
    with:
      name: mauistocktake-android-build      #A

  - name: Download iOS artifact
    uses: actions/download-artifact@v2
    with:
      name: mauistocktake-ios-build         #A
```

Now that we've got the artifacts, let's start uploading them. We'll upload the Android build first with a community provided action called `upload-google-play`. There are a few ways that you can use this action, but we're going to provide the following parameters.

REMINDER

Parameters in GitHub workflows are passed using the `with` keyword.

- **serviceAccountJsonPlainText**: This is the JSON content that we downloaded as the service account key. As it's plain text we (rather than a file) we can pass it straight from the GitHub secret.
- **packageName**: This is the bundle ID.
- **releaseFiles**: This is the path to the signed AAB file(s) we want to upload. We don't need to know the specific path as we can use a glob pattern.
- **track**: The *release track* to upload the build to (e.g., internal testing,

production, etc.).

- **status:** This can be marked as in progress for a staged release (which also requires you to provide a percentage). We'll just mark this as complete to indicate that it should be rolled out to all eligible users concurrently.

Listing 12.15 shows the step that uploads the Android build to Google Play.

Listing 12.15 The Android upload step

```
- name: Upload Singed AAB
  uses: r0adkll/upload-google-play@v1
  with:
    serviceAccountJsonPlainText: ${{ secrets.GCP_SERVICE_AC }}
    packageName: com.mildredssurfshack.MauiStockTake
    releaseFiles: ./Signed.aab
    track: internal
    status: completed
```

The next logical step is to upload the iOS build, and for this we'll use another community provided action called `upload-testflight-build`. Unlike the Android upload job, this action doesn't accept a glob pattern for the file to be uploaded, so we'll insert an interim step first that gets the path to the file and sets it as an environment variable. Listing 12.16 shows the step to add to get the iOS build file path.

Listing 12.16 Set the iOS file path as an environment variable.

```
- name: Get ipa filename
  run: echo "IPA_FILENAME=$(ls -R *.ipa)" > $GITHUB_ENV
```

We're now ready to add the final step, which is to upload the iOS build. The `upload-testflight-build` action takes the following parameters:

- **app-path:** This is the file path for the ipa archive. We'll use the environment variable we set in the previous step.
- **issuer-id:** This is the issuer ID, which we have set as a GitHub secret.
- **api-key-id:** This is the API key ID, which we have set as a GitHub secret.
- **api-private-key:** This is the private key that we got as .p8 file and have

set as a GitHub secret.

Listing 12.17 shows the upload iOS step.

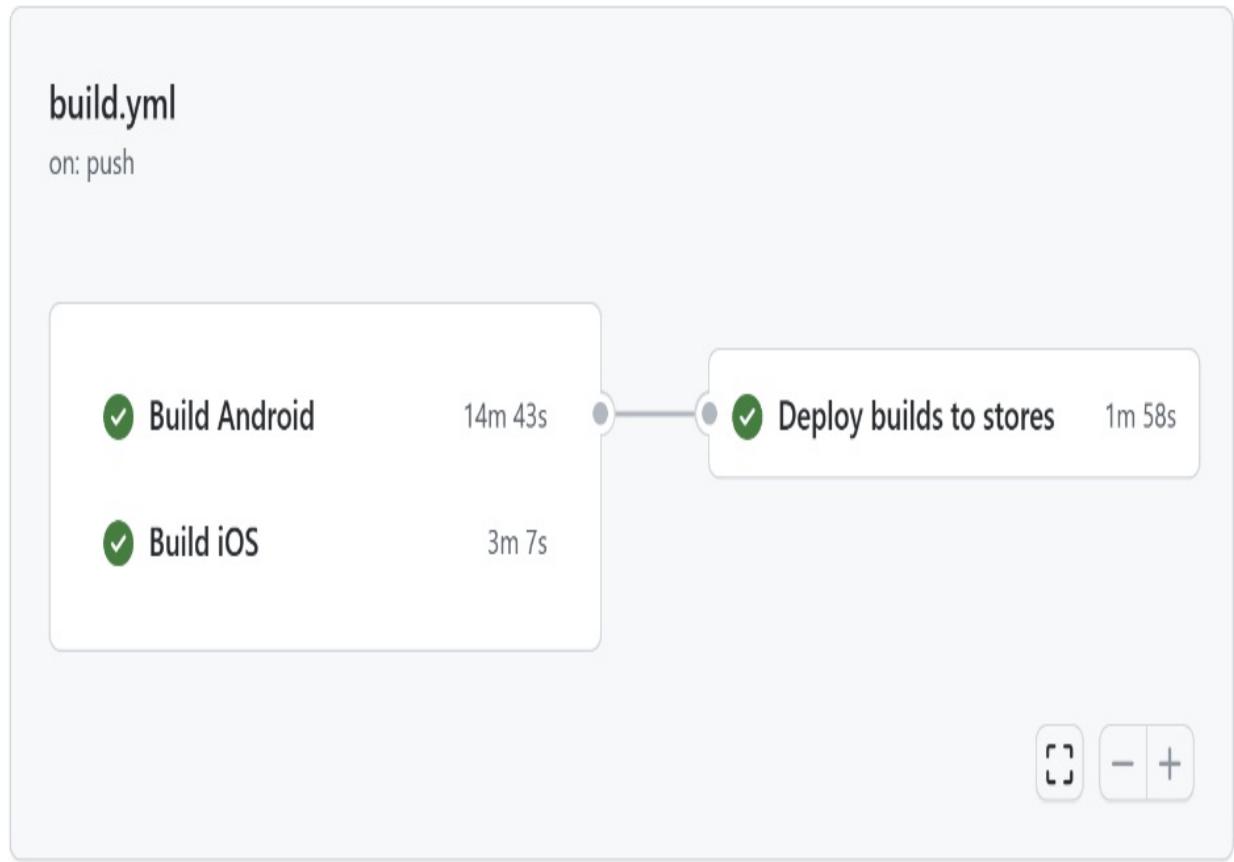
Listing 12.17 The upload iOS step

```
- name: Upload app to TestFlight
  uses: apple-actions/upload-testflight-build@v1
  with:
    app-path: ${{ env.IPA_FILENAME }}
    issuer-id: ${{ secrets.APPSTORE_ISSUER_ID }}
    api-key-id: ${{ secrets.APPSTORE_API_KEY_ID }}
    api-private-key: ${{ secrets.APPSTORE_API_PRIVATE_KEY }}
```

With the iOS upload step in place, this completes the workflow. You can find the full workflow in the chapter 12 resources folder.

At this point, you should be able to run the workflow and successfully deploy builds to the two stores. If your workflow has completed successfully, you should see some nice green ticks, as in figure 12.13.

Figure 12.13 A successfully completed workflow run shows a green tick for each job.



12.3 Next steps

At this point you've completed and end-to-end development cycle of an enterprise app. You've identified the business requirements, designed and built the app, integrated it into an enterprise architecture, and used a CI/CD pipeline to deploy the build to the stores.

This is a typical model of a real-world scenario and is not too far removed from working on an enterprise app development team (really the only thing that's missing is the agile process).

If you've enjoyed this journey, and I hope that you have, the good news is that there's still plenty left to learn. For example, in this chapter we've looked at automated build processes for Android and iOS (and the iOS process can be easily adapted for macOS). We haven't covered Windows deployment to the Microsoft store; this is because, at time of writing, automated

deployments of WinUI builds are not as reliable as I would like to be comfortable including them in this book. WinUI is a relatively new technology so some teething issues are to be expected. If you're targeting Windows desktop this is an area to keep an eye on.

There are also many features of .NET MAUI that we haven't covered (animation for example), and many that, while we have covered them, we've only scratched the surface.

But, at this stage, you've walked through the process of developing an enterprise app, and you're well equipped to pick up anything else you need as you go.

12.4 Summary

- Adding custom icons and splash screens is easy in .NET MAUI using layered SVGs. Leaving the default icon or splash screen is the hallmark of a work-in-progress, so a well-designed icon or splash screen adds professionalism and polish.
- App icons, and in fact all images in .NET MAUI, are automatically resized for each screen resolution. Using SVG images will ensure these are crisp on all devices.
- Your app is specifically identified by a bundle ID. On Windows this is a GUID, on Android, iOS and macOS it is a reverse-DNS formatted string.
- Microsoft, Apple, and Google have developer programs. You must join these to distribute your apps via their digital storefronts. Apple's developer program requires an annual membership fee. Microsoft and Google both have a one-time fee; however, Microsoft will deactivate your membership if you are not actively releasing apps or updates.
- Sign Android apps with a self-signed certificate. The Google Play developer console will expect any new versions of the app to be signed with the same certificate. Visual Studio can manage this for you, but you should export the keystore and store it securely.
- Sign iOS apps and macOS apps with a distribution certificate provided by Apple. Visual Studio can automate requesting this certificate for you. Multiple distribution certificates and profiles can be used for any app;

these are managed in the Apple Developer website.

- Using GitHub actions makes it simple to build a reusable workflow to build and deploy your apps to the Google Play store and iOS App Store.
- GitHub Actions workflows are defined in YAML files. YAML is a text format that uses indentation to denote structure and hierarchy.
- GitHub Actions workflows are divided into jobs, which are divided into steps. Workflows can define triggers that specify what causes the workflow to be run.
- The steps of a job can be shell commands (which in turn means you can execute arbitrary code), or can be pre-defined actions, provided either by GitHub or the community.
- Workflows can be run on GitHub hosted runners or self-hosted runners. Self-hosted runners are a better option for macOS jobs as macOS incurs a 10x penalty when running on GitHub.

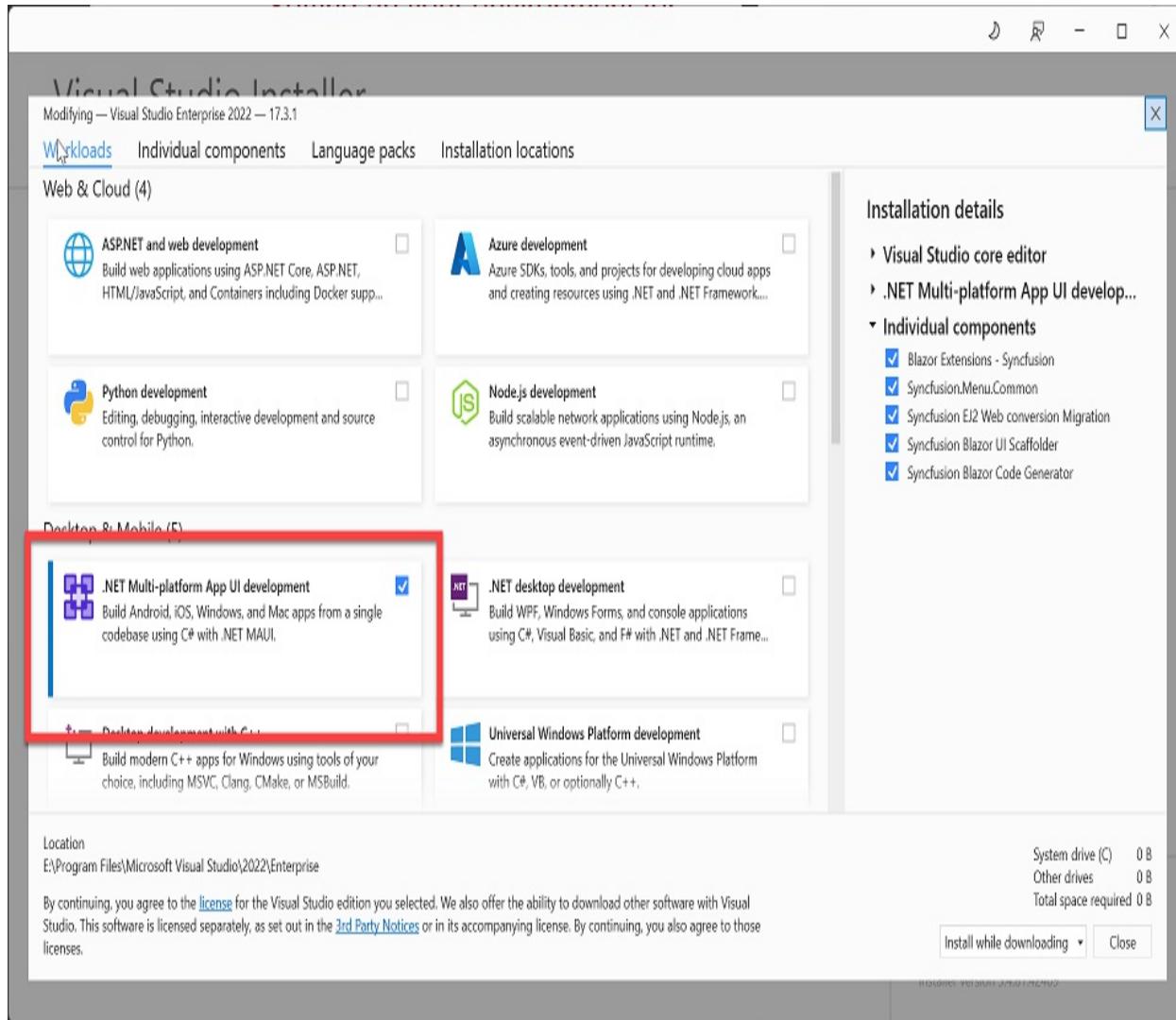
Appendix. A Setting up your environment for .NET MAUI development

Thank you for reviewing .NET MAUI in Action. When the book is complete, Appendix A will contain instructions on how to set up your environment for .NET MAUI development. As .NET MAUI has now been released, these setup instructions can now be considered final and will not change between now and when the book is released.

A.1 Setup on Windows

1. Install the latest version of Visual Studio 2022. Any edition is fine, including the free Community edition. There are no features that you would need for .NET MAUI development that are only in the paid editions.
 - a. If you already have Visual Studio installed, open the Visual Studio installer and proceed to step 2.
2. Ensure that you select the **.NET Multi-platform App UI development** workload. This will set up everything you need to develop apps with .NET MAUI

Figure A.1: Select the .NET Multi-platform App UI development workload in the Visual Studio installer



3. After you have installed Visual Studio (or added the .NET MAUI workload if you already had it installed), install the blank .NET MAUI project template by running the following command from your terminal of choice:

```
dotnet new --install Goldie.BlankMauiTemplate
```

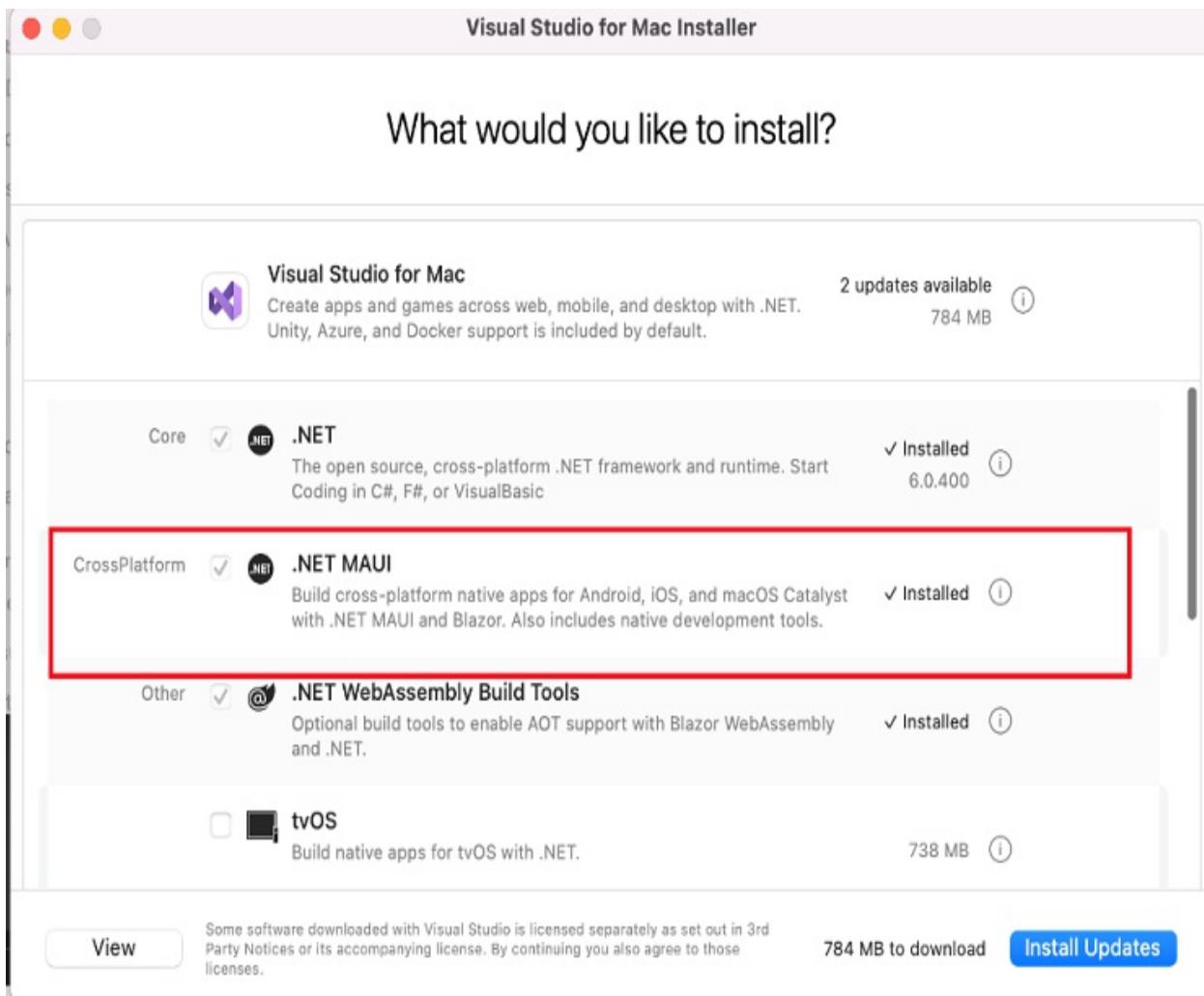
A.2 Setup for Mac

Visual Studio for Mac support for .NET MAUI is still in preview, so you must install the preview edition of Visual Studio for Mac to use the .NET MAUI workload. You can get the installer here:

<https://visualstudio.microsoft.com/vs/mac/preview/>

1. Download and run the installer.
2. Ensure you select the CrossPlatform .NET MAUI workload

Figure A.2: Select the CrossPlatform .NET MAUI workload in Visual Studio for Mac Preview installer



3. After you have installed Visual Studio (or added the .NET MAUI workload if you already had it installed), install the blank .NET MAUI project template by running the following command from your terminal of choice:

```
dotnet new --install Goldie.BlankMauiTemplate
```

A.3 Developing full stack apps locally

Running many of the apps in the book is as easy as hitting F5 (or clicking the run button, or using `dotnet run` in the CLI). There are some that talk to a REST API to retrieve data, but these APIs exist on the public internet so are easy for our apps to reach.

When you develop a full stack solution, your API will eventually be hosted on the public internet. But while you're building the app, it's much easier to run the API locally on your development machine. This is especially helpful if you're working on a full vertical slice (a complete feature that has functionality across all layers of the stack) and need to debug the API and the mobile app at the same time.

A.3.1 Full stack local development challenges

Connecting to an API running on your development machine presents two challenges. The first is how your app can access the API. When its publicly hosted it has a URL that your app can reach, but this isn't the case for your local development machine. Depending on your target framework, there are different ways you can get around this. For macOS, Windows and the iOS simulator, you can use `localhost` or the loopback IP address (`127.0.0.1`) to access your API. As the Android emulator runs a full OS, this address routes to itself, but the emulator provides a special address to access the host machine (`10.0.2.2`). If you're running your app on a physical Android or iOS device, you would have to use the IP address of your development machine on your local network, and you would also need to configure your firewall to allow external connections to your API.

The second challenge is around SSL certificates. When you run your API, you can easily instruct your machine to trust the self-signed development certificate using:

```
dotnet dev-certs https -trust
```

You can read more about ASP.NET Core developer certificates on Scott Hanselman's blog post here: <https://www.hanselman.com/blog/developing-locally-with-aspnet-core-under-https-ssl-and-selfsigned-certs>. This works well for ASP.NET Core, but getting your .NET MAUI app to trust this certificate can be tricky, and the method is different for each OS. It can also

be unreliable, and Microsoft recommend bypassing security checks for SSL certificates in your code while debugging.

Unfortunately, this last point presents a roadblock for the apps we build in this book, because we'll be using the OS's default web browser for authentication (see section 8.xx), and we can't control the browser's certificate handling behavior from our code.

A.3.2 Setting up ngrok

If you want to build and test your app on a number of different devices, rather than go through the trouble of installing the certificate each time, and changing your code to point to a different URL depending on where it's running, it's much easier to use a reverse proxy or tunnel.

These tools establish a tunnel from your development machine to a server on the internet that sends traffic back across the tunnel. This creates a publicly routable URL that you can use to access your local development machine. The most popular of these tunneling tools is ngrok as it offers a free tier and is easy to set up and use.

You don't have to use ngrok

There are plenty of alternatives to ngrok available. I use Packetriot, but have chosen to use ngrok here because Packetriot is not as easy to set up, and is not differentiated on the free tier (I have a paid subscription). Cloudflare is also a popular choice.

If you already have another reverse proxy or tunneling solution, you can skip this section and use whatever you are comfortable with.

Go to <https://ngrok.com>, and click on the **Sign up** button. After signing up for a free account, follow the download instructions for your host operating system.

Download and install ngrok, then go back to the website and click on the **Your Authtoken** tab. Follow the instructions here to add your auth token to your installed ngrok configuration.

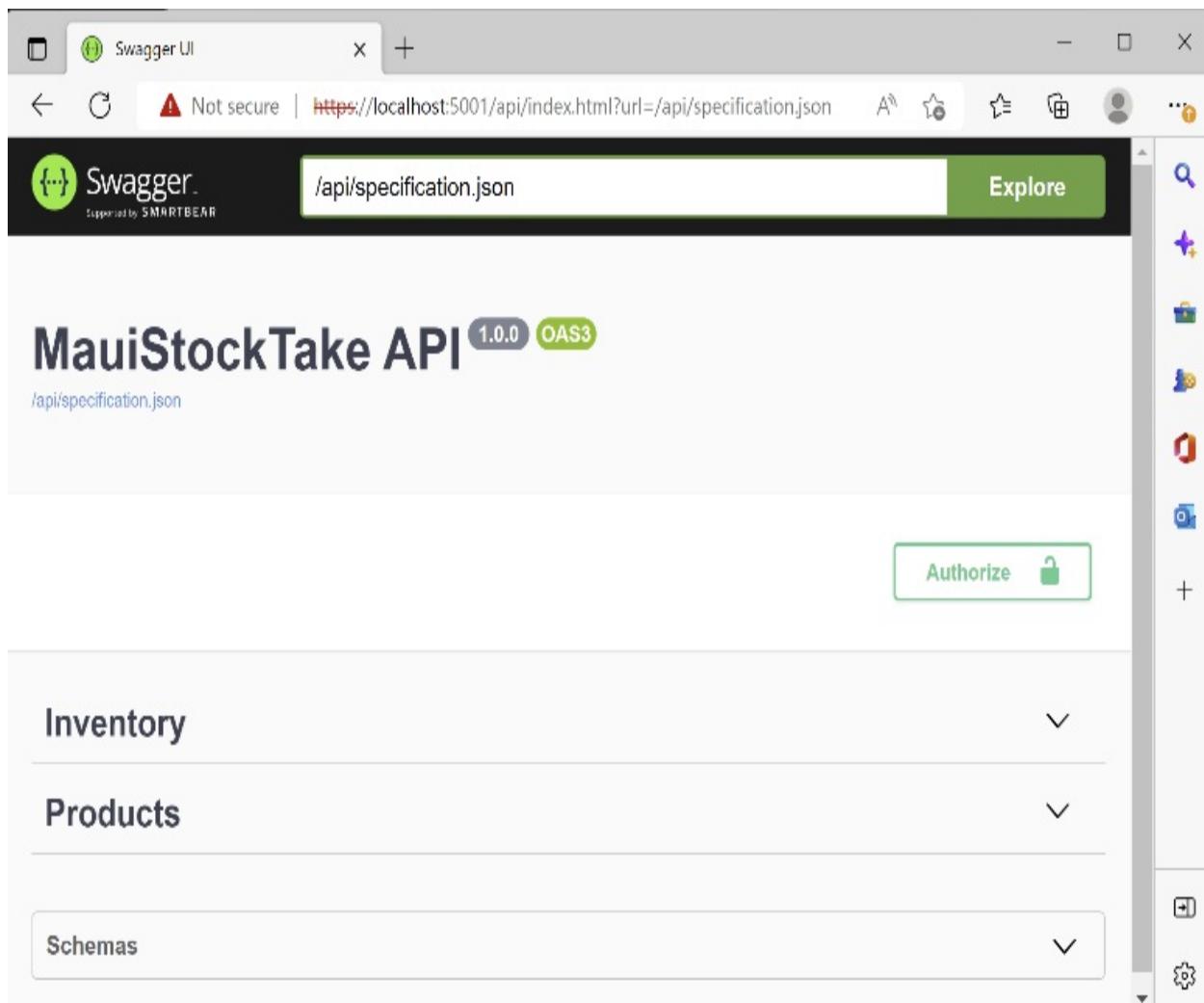
Once you've got ngrok installed and authorized, you can set up a tunnel. Run your API project and it will start up on your local machine with HTTPS running on port 5001.

Note

if you want to run on a different port, you can edit the configuration in the launchsettings.json file in the Properties folder of the API project.

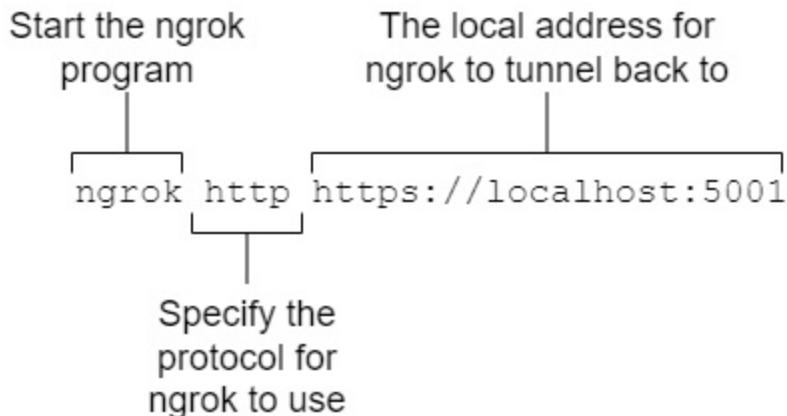
If you've got Swagger setup up (the MauiStockTake API that you will download in Chapter 7 has this) you can verify that the WebAPI project is running by opening a browser and going to the Swagger page (for MauiStockTake, this will be <https://localhost:5001/api>). This should bring up a Swagger page with information about the API endpoints and schema, similar to figure 3.

Figure A.3 The MauiStockTake API's Swagger page, running on localhost on port 5001



With your API running, you can open an ngrok tunnel by using the command shown in figure 4.

Figure A.4 The ngrok command. The first part of the command (ngrok) invokes the ngrok program, the next part (http) tells ngrok what protocol to use. The third part (<https://localhost:5001>) tells ngrok what local address to use.



Tip

ngrok is a standalone executable, so your command prompt must be at the path where the file is located. You can make life easier by copying the file to a sensible location (e.g., a folder in Program Files on Windows) and adding the path to your PATH environment variable (applicable on all platforms).

After you run the ngrok command, it will start the tunnel and show you the public URL it has generated that is tunnelling back to your local machine. See figure 5 for an example.

Figure A.5 When ngrok is running it will show you the URL it has generated that tunnels back to your local machine

```
ngrok
```

```
Hello World! https://ngrok.com/next-generation
```

Session Status	online												
Account	Matt Goldman (Plan: Free)												
Version	3.0.7												
Region	Australia (au)												
Latency	5ms												
Web Interface	http://127.0.0.1:4040												
Forwarding	https://d2bf-159-196-124-207.au.ngrok.io → https://localhost:5001												
Connections	<table><thead><tr><th>ttl</th><th>opn</th><th>rt1</th><th>rt5</th><th>p50</th><th>p90</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0.00</td><td>0.00</td><td>0.00</td><td>0.00</td></tr></tbody></table>	ttl	opn	rt1	rt5	p50	p90	0	0	0.00	0.00	0.00	0.00
ttl	opn	rt1	rt5	p50	p90								
0	0	0.00	0.00	0.00	0.00								

With ngrok running, you can paste the URL it has generated into a web browser. Append /api on the end, and you should see exactly the same Swagger page you saw in figure 8.xx

Now that ngrok is running, you can use the URL it has generated in your .NET MAUI app. You don't have to worry about SSL certificates, and the URL will work no matter what platform you're running on, whether on a physical device, an emulator or a simulator.

Appendix B. Upgrading a Xamarin.Forms app to .NET MAUI

Support for Xamarin.Forms ends on May 1st, 2024, meaning after this there will be no fixes or updates shipped (and of course no new features either). If you maintain a Xamarin.Forms app, you'll need to upgrade to .NET MAUI before then to stay in support.

Depending on the nature and complexity of your app, you may be able to reuse nearly all of your code. In other cases, you may need almost a complete rewrite of the UI (or 100%, if you're switching to MAUI Blazor). .NET MAUI is an evolution of Xamarin.Forms and shares a lot of similarities, but it's also been rewritten from the ground up, which means upgrading is not as simple as updating some package references.

In this appendix, we'll take a high-level look at the process of upgrading an app from Xamarin.Forms to .NET MAUI based on a real-world case study. At SSW, we built the SSW Rewards app in Xamarin.Forms in 2019. It has now been updated to .NET MAUI (release pending at time of writing), and in this appendix we'll examine the process involved in the upgrade.

SSW Rewards is open source, so you can see the pull request where the .NET MAUI version was merged here:

<https://github.com/SSWConsulting/SSW.Rewards.Mobile/pull/406>.

Additionally, the team are putting together a video documenting our learnings from this process. At time of writing, this video is not currently available; however it will be published on SSW's YouTube channel, which you can find here: <https://www.youtube.com/@sswtv>.

This appendix contains notes and observations of the process of upgrading the app from Xamarin.Forms to .NET MAUI. While there was undoubtedly a lot of work involved, a lot of the refactoring reduced complexity and left us with a much simpler, cleaner and easier to maintain app.

B.1 The .NET Upgrade Assistant

The .NET Upgrade Assistant is a tool designed to simplify the process of updating legacy .NET code projects to modern versions. The following links provide information about the upgrade assistant, how to install it, and some .NET MAUI specific notes:

- <https://github.com/dotnet/upgrade-assistant>
- https://github.com/dotnet/upgrade-assistant/blob/main/docs/maui_support.md
- [https://github.com/dotnet/maui/wiki/Migrating-from-Xamarin.Forms-\(Preview\)](https://github.com/dotnet/maui/wiki/Migrating-from-Xamarin.Forms-(Preview))

While the .NET Upgrade Assistant will run on your code and make changes to it, we didn't find that the end result was a usable codebase that we could work on. The .NET Upgrade Assistant updated namespaces and references, and did the bulk of the work, but we found that the process of switching from a three-project solution in Xamarin.Forms (the shared library, and the iOS and Android projects) was easier to achieve manually.

We created a fresh .NET MAUI project and used this as the basis of our updated version; however, the code that had been updated by the .NET Upgrade Assistant was invaluable, and we copied and pasted almost everything from the code that was updated by the upgrade assistant into our new .NET MAUI project.

B.2 Importing code

Following the namespace and reference updates automatically completed by the .NET Upgrade Assistant, we copied nearly all of the code from the shared project into the new .NET MAUI project. At this stage, we just brought the files across without making any changes to them. We did this in reverse dependency order:

1. Models
2. Services
3. ViewModels

4. Views

The views didn't need any changes at this stage; while we did need to make several tweaks, these were visual bugfixes rather than build errors, and came later in the process (see B.6 below). The .NET Upgrade Assistant updated the C# and XAML namespace references, replacing Xamarin.Forms namespaces with .NET MAUI namespaces, which allowed us to import the code files without further effort.

B.3 Updating to modern patterns

.NET MAUI uses modern patterns not available in Xamarin.Forms, for example the generic host builder pattern and built-in dependency injection. Because we didn't have these features in Xamarin.Forms, we had several workarounds in the legacy version of the app. For example, we used TinyIoC and a custom Resolver class. In the .NET MAUI app, we removed these and registered all dependencies with the built-in DI container. This required a fair bit of refactoring; many ViewModels and views made calls to the Resolver class, and we updated these to constructor-inject dependencies instead.

.NET MAUI also simplifies management of resources and assets. Fonts and images are significantly easier to use in .NET MAUI. In the Xamarin.Forms app, we had multiple versions of many images (one for each resolution on each platform). With .NET MAUI, we were able to eliminate these (as this is managed automatically with Resizerizer) and use a single image asset instead. In the Xamarin.Forms app, fonts were registered as a static resource, whereas in .NET MAUI they are registered in `MauiProgram` and referred to by name. This necessitated some more refactoring as we had converters for displaying icon fonts which would choose between Font Awesome for branded icons, like the Angular icon for example, and Fluent Icons for non-branded icons, like a search icon. Previously these converters returned the font resource from the app's dictionary; in the updated .NET MAUI version, they simply returned a string with the name of the font.

In the Xamarin.Forms app, we had a base service that had a protected `HttpClient` that could be used by derived services. In the .NET MAUI app, we updated this to use the `IHttpClientFactory` pattern and registered the

client in `MauiProgram`.

We also had to update the app icons and splash screen. Fortunately, this is much easier in .NET MAUI and this was a quick update using existing assets.

B.4 Updating dependencies

Many of the packages we used in the `Xamarin.Forms` app don't work in .NET MAUI. For example, a core feature of the app is QR code scanning, and in the legacy app we used `zxing.Net.Mobile.Forms` for this. This package isn't available for .NET MAUI; however, a port of the underlying ZXing library is `-ZXing.Net.Maui`, created by Jonathan Dick on the .NET MAUI team; you can find out more about it here: <https://github.com/Redth/ZXing.Net.Maui>.

While we didn't use any UI component libraries, we did have three key UI dependencies. The first was SkiaSharp, which was used to create a custom circular progress bar control. Initially, we replaced this with a new control created using the `Microsoft.Maui.Graphics` library. However, there were some bugs, and while we were able to work around these, `Microsoft.Maui.Graphics` is still listed as 'experimental', so we decided not to introduce additional technical debt, but instead revert to the SkiaSharp control with a note to replace it with `Microsoft.Maui.Graphics` once it's stable.

Microsoft.Maui.Graphics

`Microsoft.Maui.Graphics` is a new library for drawing and manipulating 2D graphics. It allows you to draw shapes and paths and render and manipulate graphics files and text. While it's included as part of .NET MAUI, it is independent and can .NET project types too.

The original plan for this book included a chapter on creating controls with `Microsoft.Maui.Graphics`; however, as it is not yet stable at time of writing, the content included too many workarounds to be of sufficient value. If you are interested in seeing the code from that chapter, you can see the `MauiBatmobile` repo in GitHub here: <https://github.com/matt-goldman/MauiBatmobile>.

To find out more about the `Microsoft.Maui.Graphics` library and how you can use it in your .NET MAUI apps, check out the official website at <https://maui.graphics>.

The second UI dependency was on Lottie, a library for displaying Lottie animation files. Lottie files are JSON files describing vector frames of an animation and is an efficient way to include animations in mobile apps. Lottie is made by AriBnB, and the library used in the `Xamarin.Forms` app, `Com.Airbnb.Xamarin.Forms.Lottie`, has not been ported to .NET MAUI; however Lottie animations are supported in the `SkiaSharp.Extended` package. We included this package in the .NET MAUI app and refactored our code for displaying Lottie animations to use this new library's syntax.

Using the built-in `DisplayAlert` API in .NET MAUI is good for simple messages or actions, but for windows or dialogs that retain in-app branding, you need something else. The third UI dependency was on a package called `Rg.Plugins.Popups`, which was used for displaying popup dialogs. When we first started working on the update, this package hadn't been ported to .NET MAUI, so we initially re-wrote all our dialogs to use `Popup` from the .NET MAUI Community Toolkit. But before we finished, a new version of `Rg.Plugins.Popups` specifically built for .NET MAUI called `Mopups` was released, so we switched to this.

The last dependency update was to remove the `CrossMedia` plugin, which we used for custom avatars (users can take a photo or select one from their gallery). This plugin isn't compatible with .NET MAUI, and .NET MAUI has built-in support for the camera and gallery anyway.

B.5 Platform specifics

Updating the platform specific elements of the app was the simplest part of the process. The platform specifics consisted of three things: app manifest (`AndroidManifest.xml` and `info.plist`), custom renderers and a web callback activity for Android (to handle OAuth redirects back to the app).

Updating the manifests was simple; we just copied over what we had in the Xamarin projects. Updating the custom renderers to use the new handlers

architecture was very straightforward too, and in fact updating to .NET MAUI made this even easier. In the Xamarin.Forms solution, we had a subclassed `Entry` control in the shared project and a custom renderer in each platform project. In the .NET MAUI app, we simply had the single subclassed control with the handler mappings in the same file.

The web callback activity was just copied from the Xamarin project to the `Platforms | Android` folder in the .NET MAUI project. The only change was to remove the reference to `Xamarin.Essentials` and convert to a file-scoped namespace.

B.6 Fixing bugs

At this stage, we had a technically working app. It would build and run, although it looked janky and there were functional and visual bugs. Some of these had to do with slight differences between Xamarin.Forms and .NET MAUI. For example, thickness values are handled differently and had to be tweaked. In other cases, the logic of how .NET MAUI renders some things is different to how Xamarin.Forms worked.

One example is scrolling views nested inside a stack layout. In Xamarin.Forms, we had `CollectionView` inside `StackLayout`. In .NET MAUI, if you place a `CollectionView` (or any scrolling view) inside a `StackLayout`, `VerticalStackLayout`, or `HorizontalStackLayout`, it doesn't scroll. This is a change in behavior from Xamarin.Forms; in .NET MAUI a stack layout can expand infinitely off-screen, so the child view doesn't detect that its height is constrained and therefore does not enable scrolling. We worked around this by updating all our scrolling views to be contained inside a `Grid` instead.

This is the final step of the update to .NET MAUI, and is continuing indefinitely. We will continue to tweak the code and fix UI bugs as we discover them. But at this stage, even though there will always be more work to do, we consider the update complete.