Introducing .NET MAUI

Build and Deploy Cross-Platform Applications Using C# and .NET 9.0 Multi-Platform App UI

Second Edition

Shaun Lawrence

Apress®

Introducing .NET MAUI

Build and Deploy Cross-Platform Applications Using C# and .NET 9.0 Multi-Platform App UI

Second Edition

Shaun Lawrence

Apress®

Introducing .NET MAUI: Build and Deploy Cross-Platform Applications Using C# and .NET 9.0 Multi-Platform App UI, Second Edition

Shaun Lawrence
St Ives, UK

ISBN-13 (pbk): 979-8-8688-1188-3 ISBN-13 (electronic): 979-8-8688-1189-0
https://doi.org/10.1007/979-8-8688-1189-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Ryan Byrnes
Editorial Project Manager: Gryffin Winkler

Cover designed by eStudioCalamar

Cover Image by hmmunoz512 from Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

This book is dedicated to Soco our beloved family dog who sadly passed before this second edition was written. Boy, you helped me throughout the first edition and I know how you always loved a good book with the girls at bedtime. Rest well my friend.

Table of Contents

About the Author xix

About the Technical Reviewer xxi

Acknowledgments xxiii

Introduction xxv

Chapter 1: Introduction to .NET MAUI 1

Abstract 1

What Is .NET MAUI? 1

Digging a Bit Deeper 4

Where Did It Come From? 5

How It Differs from the Competition 6

Why Use .NET MAUI? 7

Supported Platforms 7

Code Sharing 8

Developer Freedom 9

Community 9

Fast Development Cycle 10

Performance 10

Strong Commercial Offerings 11

Limitations of .NET MAUI 12

No WebAssembly (WASM) Support 13

No Camera API 13

Apps Won't Look Identical on Each Platform 13

Lack of Media Playback Out of the Box 14

The Glass Is Half Full, Though 15

How to Build .NET MAUI Applications 15

Visual Studio 15

Rider 16

Visual Studio Code 16

Summary 17

Chapter 2: Building Our First Application 19

Abstract 19

Setting Up Your Environment 20

macOS 20

Windows 25

Visual Studio to macOS 27

Troubleshooting Installation Issues 31

.NET MAUI Workload Is Missing 31

Creating Your First Application 33

Creating in Visual Studio 33

Creating in the Command Line 36

Building and Running Your First Application 37

Getting to Know Your Application 39

WidgetBoard 40

Summary 41

Source Code 41

Chapter 3: The Fundamentals of .NET MAUI 43

Abstract..43

Project Structure 43

/Platforms/ Folder...46

/Resources/ Folder 49

Where to Begin?...54

Generic Host Builder 55

What Is Dependency Injection? 55

Registering Dependencies...59

Application Lifecycle 61

Application States...61

Lifecycle Events...62

Handling Lifecycle Events...63

Cross-Platform Mappings to Platform Lifecycle Events 65

Platform-Specific Lifecycle Events..66

Summary..71

Chapter 4: An Architecture to Suit You 73

Abstract..73

A Measuring Stick...74

Prerequisites 74

Model View ViewModel (MVVM) 75

Model..76

View 77

ViewModel 78

Model View Update (MVU)...80

Getting Started with MauiReactor 81

Overview of the MVU Project Format.....................................82

Adding Your MVU Implementation 83

XAML vs. C# Markup 85
Plain C# 86
C# Markup 87
Chosen Architecture for This Book 88
Adding the ViewModels 89
Adding Views 94
Viewing Your Widget 97
MVVM Enhancements 99
Summary 103
Source Code 104
Chapter 5: User Interface Essentials 105
Abstract 105
Prerequisites 105
Models 106
Pages 107
ViewModels 108
App Icons 109
Adding Your Own Icon 110
Platform Differences 111
Splash Screen 112
XAML 113
Dissecting a XAML File 114
Building Your First XAML Page 116
Layouts 118
AbsoluteLayout 118
FlexLayout 121
Grid 122

HorizontalStackLayout....125
VerticalStackLayout....126
Data Binding....129
Binding....129
Applying the Remaining Bindings....134
MultiBinding....135
Command....138
Compiled Bindings....141
Make Use of the BoardDetailsPage....142
Taking Your Application for a Spin....143
Summary....144
Source Code....144
Chapter 6: Shell....145
Abstract....145
Prerequisites....145
Pages....145
ViewModels....147
Shell....149
ShellContent....150
Navigation....151
Flyout....157
Tabs....166
Search....172
ToolbarItems....177
Add a ToolbarItem to a ContentPage's ToolbarItems....178
Changing the PresentationMode of a ContentPage....179

Summary....187
Source Code....188
Extra Assignment....189
Source Code....189
Chapter 7: Creating Our Own Layout 191
Abstract....191
Placeholder....192
ILayoutManager....194
BoardLayout....195
BoardLayout.xaml....196
BoardLayout.xaml.cs....198
FixedLayoutManager....203
Accepting the Number of Rows and Columns for a Board....205
Providing Tap/Click Support Through a Command....207
Building the Board Layout....209
Setting the Correct Row/Column Position for Each Widget....211
Using Your Layout....213
Adding a Factory That Will Create Instances of Your Widgets....213
WidgetTemplateSelector....218
Updating FixedBoardPageViewModel....220
Finally Using the Layout....222
Summary....224
Source Code....224
Extra Assignment....225
Source Code....225

Chapter 8: Accessibility....227
Abstract....227
What Is Accessibility?....227
Why Make Your Applications Accessible?....228
What to Consider When Making Your Applications Accessible....228
How to Make Your Application Accessible....229
Screen Reader Support....230
Suitable Contrast....238
Dynamic Text Sizing....240
Testing Your Application's Accessibility....245
Android....245
iOS....245
macOS....246
Windows....246
Useful Resources....247
Accessibility Checklist....247
A Guide for Making Apps Accessible....248
Summary....249
Source Code....250
Extra Assignment....250

Chapter 9: Advanced UI Concepts....251
Abstract....251
Adding the Ability to Add a Widget to a Board....252
Possible Ways of Achieving Your Goal....252
The Chosen Approach....254
Styling....261
Examining the Default Styles....263

Creating a Style 265
AppThemeBinding 267
Further Reading 268
Triggers 268
Creating a DataTrigger 269
EnterActions and ExitActions 270
Creating a TriggerAction 271
Further Reading 273
Basic Animations 274
Combining Basic Animations 275
Cancelling Animations 276
Easings 277
Complex Animations 278
Combining Triggers and Animations 283
Behaviors 284
Creating Our Behavior 285
Attaching Our Behavior 287
Taking the Application for a Spin 288
Fonts 290
Embed the Font 291
Configure the Font 291
Use the Font 291
Taking the Application for a Spin 291
Summary 292
Source Code 294
Extra Assignment 294
Animate the BoxView Overlay 294

Animate the New Widget 294
Source Code 294

Chapter 10: Local Data 295

Abstract 295
What Is Local Data? 295
File System 296
Cache Directory 296
App Data Directory 297
Database 298
Repository Pattern 299
SQLite 311
LiteDB 319
Database Summary 326
Application Settings (Preferences) 327
What Can Be Stored in Preferences? 327
Setting a Value in Preferences 328
Getting a Value in Preferences 329
Checking If a Key Exists in Preferences 332
Removing a Preference 332
Displaying Our Preferences 332
Secure Storage 336
Storing a Value Securely 338
Reading a Secure Value 339
Removing a Secure Value 339
Platform Specifics 339
Viewing the Result 341

Summary....343
Source Code....344
Extra Assignment....345
Source Code....345
Chapter 11: Remote Data....347
Abstract....347
What Is Remote Data?....347
Considerations When Handling Remote Data....348
Web Services....350
The Open Weather API....350
Adding Some State....369
Simplifying Web Service Access....378
Prebuilt Libraries....379
Code Generation Libraries....379
Further Reading....381
StateContainer from CommunityToolkit.Maui....381
Summary....382
Source Code....383
Extra Assignment....383
TODO Widget....383
Quote of the Day Widget....383
NASA Space Image of the Day Widget....384
Source Code....384
Chapter 12: Getting Specific....385
Abstract....385
.NET MAUI Essentials....385
Permissions....386

Using the Geolocation API 391
Configuring Platform-Specific Components 397
Platform-Specific API Access 403
Platform-Specific Code with Compiler Directives 403
Platform-Specific Code in Platform Folders 405
Overriding the Platform-Specific UI 407
OnPlatform 407
Handlers 410
Summary 417
Source Code 417
Extra Assignment 418
Barometer Widget 418
Geocoding Lookup 418
Source Code 418
Chapter 13: Testing 419
Abstract 419
Unit Testing 419
Unit Testing in .NET MAUI 420
Adding Your Own Unit Tests 421
Testing Your View Models 426
Testing Asynchronous Operations 428
Testing Your Views 436
Device Testing 438
Creating a Device Test Project 438
Adding a Device-Specific Test 441
Running Device-Specific Tests 443

Snapshot Testing..444
Snapshot Testing Your Application...445
Passing Thoughts 449
Summary...449
Source Code 450
Chapter 14: Automation Testing 451
Abstract...451
What Is Automation Testing?...451
Automation Testing in .NET MAUI...452
Installing Appium 453
Creating the Automation Test Project...459
Add the Appium NuGet Package...461
Creating an Appium Server...461
Creating the Appium Platform Drivers..463
Parameterizing the Tests 469
Writing the Automation Tests 472
Testing the Add New Board Button..473
Adding a Test to Create Boards 476
Adding a Test to Interact with a CollectionView 478
Summary...479
Source Code 480
Chapter 15: Let's Get Graphical 481
Abstract...481
.NET MAUI Graphics 481
Drawing on the Screen...482
Further Reading..485

Building a Sketch Widget 485
Creating the SketchWidgetViewModel 485
Representing a User Interaction 486
Creating the SketchWidgetView 487
Registering Your Widget 492
Taking Your Widget for a Test Draw 492
Building an Analog Clock Widget 493
Creating the AnalogClockWidgetView 494
Creating the AnalogClockWidgetViewModel 497
Registering Your Widget 502
Taking Your Widget for a Test Draw 502
Summary 503
Source Code 504
Extra Assignment 504
Source Code 504

Chapter 16: Releasing Our Application 505

Abstract 505
Distributing Your Application 506
macOS 506
Windows 507
Android 507
iOS 511
macOS 516
Windows 517
Optimizing Your Application 519
Following Good Practices 520
Performance 521

Trimming 524
Ahead-of-Time Compilation 531
When Libraries Don't Support Trimming or AOT 536
Results 537
Crashes/Analytics 537
Obfuscation 538
Distributing Test Versions 541
Summary 541

Chapter 17: Conclusion 543
Abstract 543
Looking at the Final Product 543
Taking the Project Further 546
Useful Resources 547
StackOverflow 547
GitHub 548
YouTube 548
Social Media 548
Yet More Goodness 549
Looking Forward 549

Index 551

About the Author

Shaun Lawrence is an experienced software engineer who has been specializing in building mobile and desktop applications for the past 20 years. He is a recognized Microsoft MVP in Development Technologies for his work helping the community learn and build with Xamarin.Forms and .NET MAUI. His recent discovery of the value he can add by sharing his experience with others has thrust him on to the path of wanting to find any way possible to continue it.

Shaun actively maintains several open source projects within the .NET community. A key project for the scope of this book is the .NET MAUI Community Toolkit where he predominantly focuses on building good quality documentation for developers to consume. Shaun lives in the UK with his wife, two children, and their dog.

About the Technical Reviewer

Gerald Versluis is a Senior Software Engineer at Microsoft working on .NET MAUI. Since 2009, Gerald has been working on a variety of projects, ranging from front end to back end and anything in between that involve C#, .NET, Azure, ASP.NET, and all kinds of other .NET technologies. At some point, he fell in love with cross-platform and mobile development with Xamarin, now .NET MAUI. Since that time, he has become an active community member, producing content online and presenting about all things tech on conferences all around the world.

Acknowledgments

I have a few people that I would like to thank for their assistance.

Firstly, Gerald: Not only have you reviewed this book, but you have been there to help me overcome some tricky obstacles with either one of your many YouTube videos or just general experience to guide me to a sensible solution.

Secondly, Bailey, our family cockerpoo: Forcing me out on those lengthy walks come rain or shine really helped me to clear my head and provide some time for my brain to catch up. I can't tell you how many solutions we came up with and had to rush home to jot them down!

Thirdly, the team at Apress: You have all helped me to keep on track and provide great support whenever I got stuck.

Finally, my family – my wife Levinia and daughters Zoey and Hollie: Without your encouragement I would not have been brave enough to take on the challenge to write this book. I am so grateful for the help and sacrifices you have each made to help me get this book finished and even the little slots of time waiting for you to finish dance class.

Introduction

Welcome to *Introducing .NET MAUI*.

This book has been written for developers who are new to .NET MAUI and cross-platform development. You should have basic knowledge of C# but require no prior knowledge of using .NET MAUI. The content ranges from beginner through to more advanced topics and is therefore tailored to meet a wide range of experiences. In fact, my intention is to allow you to learn different levels of content upon multiple reads of this book; you can feel free to skip past the more complex scenarios and just apply the results at the end of the chapter if you do not feel ready for it. Then upon subsequent read-throughs, I expect more and more to make sense.

This book provides an in-depth explanation of each key concept in .NET MAUI, and you will then use these concepts in practical examples while building a cross-platform application. The content has been designed to primarily flow with the building of this application; however, there is a secondary theme that involves grouping as many related concepts as possible. The idea behind this is to both learn as you go and to have content that closely resembles reference information, which makes returning to this book as easy as possible.

All code examples in this book, unless otherwise stated, are applied directly to the application you are building. Once key concepts have been established, the book will offer improvements or alternatives to simplify your future experiences as you build production-worthy applications. This book does not rely upon these simplifications for all the practical examples, and the reason for this is simple: I strongly believe that you need to understand the concepts before you start to use them or use libraries that do it for you.

The application that we will build together throughout the course of this book will be created using the default template provided by .NET MAUI. This means that the project will have Nullable Reference Types turned on; therefore, I would strongly recommend reading up on them if you are unfamiliar before undertaking this book. Microsoft provides some good documentation at `https://learn.microsoft.com/dotnet/csharp/nullable-references`. Also note that we will aim to build an application that is free of build warnings; don't worry too much at this early stage, but you will gain an understanding of why this is extremely important by the time we reach the end of this book.

Finally, all chapters that involve adding code into the application project contain a link to the resulting source code repository. This is to show the final product and for you to use as a comparison if anything goes wrong during your building of the application.

CHAPTER 1

Introduction to .NET MAUI

Abstract

In this chapter, you will gain an understanding of what exactly .NET MAUI is, how it differs from other frameworks, and what it offers you as a developer wishing to build a cross-platform application that can run on both mobile and desktop environments. I will also cover the reasons why you should consider it for your next project by weighing the possibilities and limitations of the framework as well as the rich array of tooling options.

What Is .NET MAUI?

.NET Multi-platform App UI, or .NET MAUI for short, is a cross-platform framework that allows developers to build mobile and desktop applications primarily written in C# and XAML. It allows developers to target both mobile (Android and iOS) and desktop (macOS and Windows) platforms from a single code base. Figure 1-1 shows the platforms officially supported by .NET MAUI and Microsoft.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_1

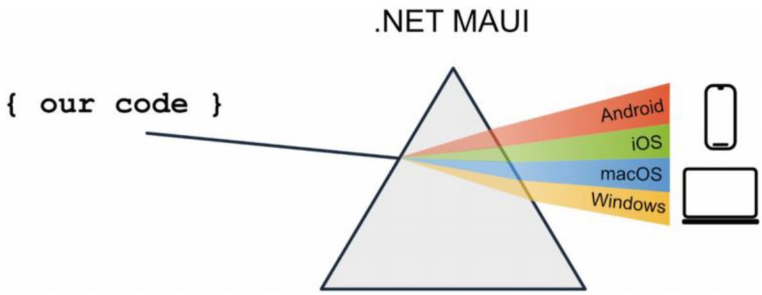

Figure 1-1. *.NET MAUI platform support*

.NET MAUI provides a single API that allows developers to write code once and run it anywhere. When building a .NET MAUI application, you write code that interacts with this single cross-platform API, and .NET MAUI provides the bridge between your code and the platform-specific layer.

If you take a look inside the prism in Figure 1-1, you can start to understand the components that .NET MAUI both uses and offers. Figure 1-2 shows how an Android application is compiled. We can make the statement that when compiling our application for Android, **Your code** is compiled against **.NET MAUI** and in turn **.NET for Android**.

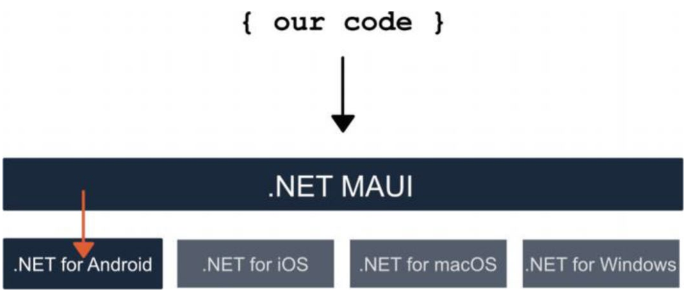

Figure 1-2. *Interacting with .NET MAUI APIs*

Figure 1-2 shows how our code only directly makes use of the .NET MAUI APIs, but then under the hood, .NET MAUI makes use of the .NET for Android APIs. It is through this approach that we as developers can make the most of code sharing – by making the most of the API surface that is provided to us by .NET MAUI.

There will be times when the API surface of .NET MAUI does not provide everything that you need; for this, you will need to directly access a platform feature. .NET MAUI provides enough flexibility that you can achieve this by interacting directly with the platform-specific APIs:

- .NET for Android
- .NET for iOS
- .NET for macOS
- Windows UI Library (WinUI) 3

Figure 1-3 shows how the code bypasses the .NET MAUI APIs and interacts directly with the .NET for Android APIs.

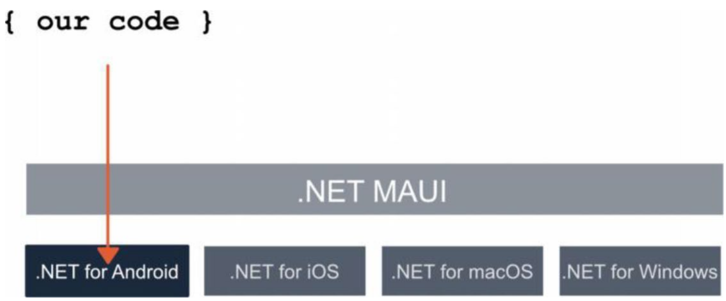

Figure 1-3. *Interacting with platform-specific APIs*

This book focuses on building applications with .NET MAUI; we have covered how .NET MAUI is built on top of the .NET frameworks for each platform (e.g., .NET for Android). This means that if you wished to only ever build an application to target Android, you can do that through the .NET for Android framework. This allows you to still build an application that targets Android and make use of .NET and C#.

Digging a Bit Deeper

There are some extra steps that the tooling will perform under the hood to get your application built and ultimately ready for use on each of the possible platforms.

When building a .NET application, even if it is not using .NET MAUI, you will very likely hear the term BCL, which is short for the *base class library*. This is the foundation of all .NET applications, and in the same way that .NET MAUI abstracts away the platforms you wish to build for, the BCL abstracts away what that platform implements when your application runs.

To run your application on your desired platform, you need a .NET runtime. For Android, iOS, and macOS, this is the Mono runtime. The Mono runtime provides the ability to run .NET code on many different platforms. For Windows, this is .NET CoreCLR. Each of these runtimes provides the functionality required for the BCL and therefore a consistent working environment across all supported platforms. I have opted to avoid telling the history of the Mono runtime, not because I don't believe it is important – without it, this book wouldn't be possible – but I feel many others have told it well already and I don't want to distract from the topic at hand.

I like to think of the BCL as the contract between what we are compiling against and what we are running that compiled code with.

Figure 1-4 shows all of the layers involved in compiling and running a .NET MAUI application.

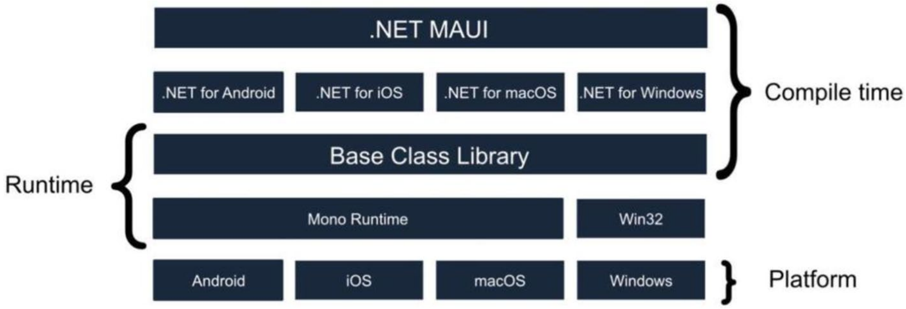

***Figure 1-4.** The full breakdown*

To continue with the example of building for Android in the previous diagrams and taking note of the diagram in Figure 1-4, the following can be said.

Your code is compiled against **.NET MAUI**, **.NET for Android**, and the **base class library**. It then runs on the **Mono runtime**, which provides a full implementation of the **base class library** on the **Android** platform.

Looking at the above statement, you can replace the parts that are platform specific with another platform (e.g., swapping Android for iOS) and the statement will still be true.

Where Did It Come From?

.NET MAUI is the evolution of Xamarin.Forms, which itself has a rich history of providing developers with a great way to build cross-platform applications. Of course, no framework is perfect, and Xamarin.Forms certainly had its limitations. Thankfully the team at Microsoft decided to take the pragmatic approach of taking a step back and evaluating all the pain points that existed for themselves as maintainers and (more importantly) for us as developers using the framework.

Not only do we therefore gain improvements from the Xamarin framework as part of this evolution, but we also get all the goodies that come with .NET such as powerful built-in dependency injection, better performance, and other topics that I will touch on throughout this book. This makes me believe that this mature cross-platform framework has finally become a first-class citizen of the .NET and Microsoft ecosystems. I guess the clue is in the first part of its new name.

On the topic of its name, .NET MAUI implies that it is a UI framework, and while this is true, this is not all that the framework has to offer. Through the .NET and the .NET MAUI platform APIs, we are provided with ways of achieving common application tasks such as file access, accessing media from the device gallery, using the accelerometer, and more. The .NET MAUI platform APIs were previously known as Xamarin Essentials, so if you are coming in with some Xamarin Forms experience, they should feel familiar but note that they have evolved to fit within .NET MAUI. I will touch on much more of this functionality as you progress through this book with the key chapters being Chapters 10 and 12.

How It Differs from the Competition

.NET MAUI provides its own abstractions of types like controls (e.g., a `Button`) and then maps them to the relevant implementation on each platform. To continue with the example of a button, this is a `UIButton` from `UIKit` on iOS and macOS, an `AppCompatButton` from `AndroidX.AppCompat.Widget` on Android, and a `Button` from `Microsoft.UI.Xaml.Controls` on Windows. Figure 1-5 shows how a .NET MAUI `Button` control is mapped to each platform-specific implementation.

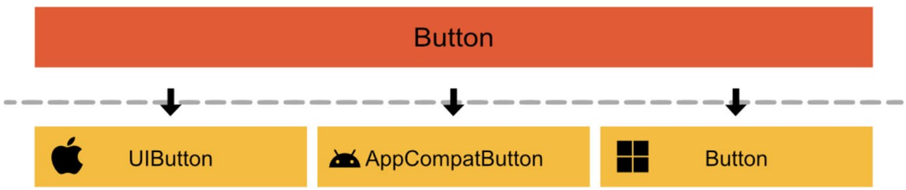

Figure 1-5. *How a Button control is mapped to the platform-specific implementations*

This gives a great level of coverage in terms of providing a common implementation that works across all platforms. With the introduction of the .NET MAUI handler architecture (which we will be looking at in more detail in Chapter 12), we truly gain the power to tweak the smallest of implementation details on a per-platform basis. This is especially useful when the common API provided by .NET MAUI may be limited down to the least amount of crossover between each platform and doesn't provide everything we need. It is worth noting that your application will render differently on each platform as it utilizes the platform-specific controls and therefore their look and feel.

Other frameworks such as Flutter opt to render their own types directly rather than mapping across to the implementations provided by each platform. These frameworks provide a common look and feel across each platform. This is a hotly contested topic, but I personally believe that making applications fit in with the platform they are running on is a big benefit.

Why Use .NET MAUI?

There are several reasons why you should consider using .NET MAUI for your next application: a large number of supported platforms, increased code sharing capabilities, an emphasis on allowing developers to build applications that fit their style, great performance, and many more. Let's take a look at them.

Supported Platforms

.NET MAUI provides official support for all of the following platforms:

- Android 5.0 (API level 21) and above
- iOS 12.2 and above

- macOS 12 and above (using Mac Catalyst) **
- Windows 11 and Windows 10 version 1809 desktop and above

*** Mac Catalyst allows native Mac apps to be built and share code with iPad apps. This is an Apple technology that allows developers to share code between Mac and iPad. For further reference, go to the Apple documentation at* https://developer.apple.com/mac-catalyst/.

.NET MAUI provides community-driven support for Tizen – the implementation is provided by Samsung so while it isn't directly provided by Microsoft, Samsung is no small company.

I thoroughly recommend checking out the documented list of supported platforms in case it has changed since the time of writing. The list can be found at https://learn.microsoft.com/dotnet/maui/supported-platforms.

Code Sharing

A fundamental goal of all cross-platform frameworks is to enable developers to focus on achieving their main goals by reducing the effort required to support multiple platforms. This is achieved by sharing common code across all platforms. Where I believe .NET MAUI excels over alternative frameworks is in the first four characters of its name; Microsoft has pushed hard to produce a single .NET that can run anywhere.

Being a full stack developer myself, I typically need to work on web-based back ends as well as mobile applications; .NET allows me to write code that can be compiled into a single library. This library can then be shared between the web and client applications, further increasing the code sharing possibilities and ultimately reducing the maintenance effort.

I have given talks based on a mobile game (www.superwordsearch.com) I built using Xamarin.Forms with a friend, where I boasted that we were able to write 96% of our code in our shared project. We have recently migrated from Xamarin.Forms to .NET MAUI and can confirm that the shared code percentage has now increased to 99%!

There are further possibilities for sharing code between web and client, such as the use of .NET MAUI Blazor Hybrid, which provides the use of web-based technologies inside a .NET MAUI application. While I won't be covering .NET MAUI Blazor Hybrid in detail in this book, Microsoft does provide some really great documentation and guides on what it is and how to build your first application with the technology at https://learn.microsoft.com/aspnet/core/blazor/hybrid/tutorials/maui. The team has also built a full and free workshop template designed for all levels of developers to work through at https://aka.ms/blazor-hybrid-workshop.

Developer Freedom

.NET MAUI offers many ways to build the same thing. Where Xamarin.Forms was largely designed to support a specific application architecture (such as MVVM, which I will talk all about in Chapter 4), .NET MAUI is different. One key benefit of the rewrite by the team at Microsoft is it now allows the use of other architectures such as MVU (Chapter 4). This allows us as developers to build applications that suit our preferences, from different architectural styles to different ways of building UIs and even different ways of styling an application.

Community

Xamarin has always had a wonderful community. From bloggers to open source maintainers, there is a vast amount of information and useful packages available to help any developer build a great mobile application. One thing that has really struck me is the number of Microsoft employees who are part of this community; they are clearly passionate about the technology and dedicate their own free time to contributing to this community. The evolution to .NET MAUI brings this community with it; Chapter 17 includes a set of resources that make discovering members within the community and guidance on how to get involved.

Fast Development Cycle

.NET MAUI offers two great ways to boost a developer's productivity.

.NET Hot Reload

.NET Hot Reload allows you to modify your managed source code while the application is running, without the need to manually pause or hit a breakpoint. Then, your code edits can be applied to your running app without the need to recompile. It is worth noting that this feature is not specific to .NET MAUI but is yet another great example of all the goodness that comes with the framework being part of the .NET ecosystem.

XAML Hot Reload

XAML Hot Reload allows you to edit the UI in your XAML files, save the changes, and observe those changes in your running application without the need to recompile. This is a fantastic feature that really shines when you need to tweak some controls.

Performance

.NET MAUI applications are compiled into native packages for each of the supported platforms, which means that they can be built to perform well.

Android has always been the slowest platform when dealing with Xamarin.Forms, and the team at Microsoft has been working hard and showing off the improvements. The team has provided some really great resources in the form of blog posts covering the progress that has been made to bring the startup times of Android applications to well below one second. These posts cover metrics plus tips on how to make your applications really fly.(https://devblogs.microsoft.com/dotnet/dotnet-9-performance-improvements-in-dotnet-maui/)

Android apps built using .NET MAUI compile from C# into intermediate language (IL), which is then just-in-time (JIT) compiled to a native assembly when the app launches.

iOS and macOS apps built using .NET MAUI are fully ahead-of-time (AOT) compiled from C# into native ARM assembly code.

Windows apps built using .NET MAUI use Windows UI Library (WinUI) 3 to create native apps that target the Windows desktop.

Strong Commercial Offerings

There are several commercial options that provide additional UI elements and other integrations such as Office document editing or PDF viewing in your .NET MAUI applications. Some options (at the time of writing) are

- Syncfusion

  "The feature-rich/flexible/fast .NET MAUI controls for building cross-platform mobile and desktop apps with C# and XAML"

  www.syncfusion.com/maui-controls

- Telerik UI for .NET MAUI

  "Kickstart your multiplatform application development with a Preview version of Telerik UI for .NET MAUI controls!"

  www.telerik.com/maui-ui

- DevExpress

  "Our .NET Multi-platform App UI Component Library ships with high-performance UI components for Android and iOS mobile development (WinUI desktop support is coming in 2022). The library includes a Data Grid, Chart, Scheduler, Data Editors, CollectionView, Tabs, and Drawer components."

  www.devexpress.com/maui/

- Grial UI Kit

  "Grial offers a set of beautiful XAML UI pages, templates, controls and helpers made for Xamarin.Forms and .NET MAUI. These cover the most typical Mobile UI patterns and are crafted by developers, for developers."

  https://grialkit.com/

Note that while these are commercial products, several of them provide free licenses for smaller companies or independent developers so I recommend checking out their products.

Limitations of .NET MAUI

I hope this doesn't get me in too much trouble with the wonderful team over at Microsoft ☺. This section is not aimed at slating the technology (I wouldn't be writing a book about something I didn't believe in); it is purely aimed at making clear what cannot be achieved or at least what is not provided out of the box, to help you as a reader best decide whether this is the right technology for your next project. Of course, I hope it is, but let's look at what I feel are its main limitations.

No WebAssembly (WASM) Support

.NET MAUI does not provide support for targeting WebAssembly. This means that you cannot target the web directly from a .NET MAUI project, but you can still run Blazor inside your .NET MAUI application. This opens the door for further code sharing; as discussed earlier, it is entirely possible to build Blazor components that can be shared between .NET MAUI Blazor Hybrid and .NET Blazor applications using the *.NET MAUI Blazor Hybrid and Web App* template.

If you do require direct WASM support, then a good alternative to .NET MAUI is the Uno Platform.

No Camera API

This has been a pain point for a lot of developers throughout the life of Xamarin.Forms and continues to be an initial pain point for .NET MAUI. There are some good arguments as to why it hasn't happened. Building a camera API against the Android Camera offering has not been an easy task, as I am sure most developers who have embarked on that journey can attest to. The sheer fact that Google has recently rewritten the entire API for a third time shows the inherent challenges.

Apps Won't Look Identical on Each Platform

Controls in .NET MAUI make use of the platform implementations; therefore, an entry control on iOS will render differently to one on Android. This approach quite often divides opinions – applications will look and feel common to the platform that they are running on. Figure 1-6 shows how a simple user interface consisting of Entry fields, a multi-line Editor, and a Button renders on each of the supported platforms.

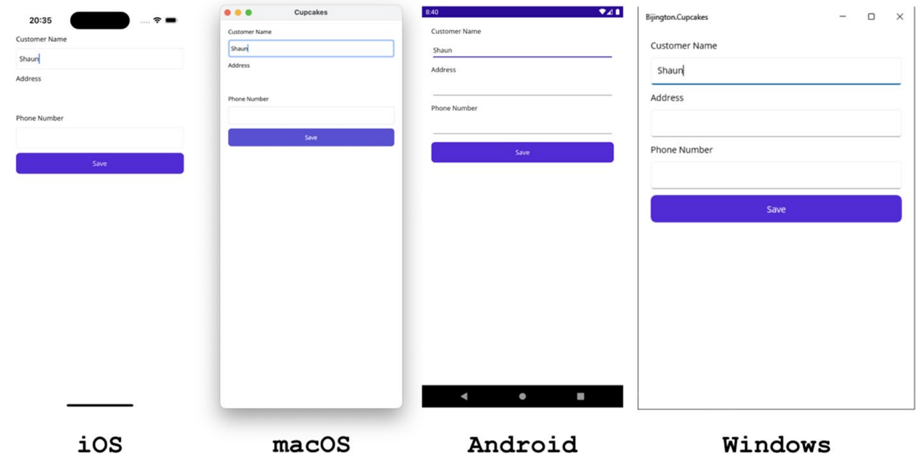

Figure 1-6. *Simple user interface renders on each of the supported platforms*

There are investigations into providing a way to avoid this and have controls render exactly the same on all platforms, but this is still at an early stage.

Lack of Media Playback Out of the Box

Playing media has become a very common task. So many apps these days offer the ability to play video or audio. I suspect this is also due to the vast differences between platforms in how they provide playback support.

While this functionality is not officially provided by .NET MAUI, this does not mean the functionality is not possible.

The Glass Is Half Full, Though

I believe that limitations are not a bad thing. Doing some things well is a far better achievement than doing everything badly! I expect the list of limitations will reduce as .NET MAUI matures. Thanks to the fact that .NET MAUI is open source, we as consumers have the ability to suggest improvements and even provide them directly to further enhance this framework. I must also add that the .NET MAUI Community Toolkit is great (of course, I am biased because I currently help to maintain it). It provides value for the .NET MAUI community, and it is also maintained by that community. Another huge advantage is that concepts in this toolkit can and have been promoted to .NET MAUI itself.

The .NET MAUI Community Toolkit also offers both APIs for interacting with a camera connected to a device and media playback. This gives me hope that one day there will be a solid choice for both camera and media playback APIs in .NET MAUI.

How to Build .NET MAUI Applications

There are several different ways to build an application with .NET MAUI. I will look at each in turn, covering some details that will hopefully help you decide which is the best fit for you.

Visual Studio

Visual Studio is a comprehensive integrated development environment or IDE that provides a great development experience. I have been using this tool for years and I can happily say that it continues to improve with each new version.

To build .NET MAUI apps, you must use at least Visual Studio 2022.

In Visual Studio (Windows), it is possible to build applications that target

- Android
- iOS*
- Windows

**A networked Mac with Xcode 13.0 or above is required for iOS development and deployment. This is due to limitations in place by Apple.*

Note that Visual Studio comes with three different pricing options, but I would like to draw your attention to the Community edition, which is free for use by small teams and for educational purposes. In fact, everything in this book can be achieved using the free Community edition.

Rider

JetBrains Rider is an impressive cross-platform IDE that can run on Windows, macOS, and Linux. JetBrains has a history of producing great tools to help developers achieve their goals. One highlight is ReSharper, which assists with inspecting and analyzing code. With Rider, the functionality provided by ReSharper is built in.

JetBrains offers Rider for free but only for educational use and open source projects.

I will be using Rider as I build the application alongside this book.

Visual Studio Code

Visual Studio Code is a very popular lightweight code editor also provided by Microsoft. Using the .NET MAUI extension and the .NET CLI, it is entirely possible to build .NET MAUI applications using no other tools. It is worth noting that while the tooling within Visual Studio Code does improve with each release, I find it leaves me feeling like I am missing out some of the great pieces of functionality that come from a fully fledged IDE. If you are new to development or new to .NET MAUI, I would thoroughly recommend using Visual Studio or Rider.

You may find references to Visual Studio for Mac online; sadly this tool has been recently discontinued; therefore, it doesn't make it into this section officially.

Summary

Throughout the course of this book, you will primarily be using Visual Studio as the tool to build your application. I will refer to Rider and Visual Studio Code in the later parts when I cover how to deploy and test macOS applications.

In this chapter, you have learned the following:

- What .NET MAUI is
- What it offers and what it does not offer
- Reasons why you should consider using it
- The tooling options available to build a .NET MAUI application

In the next chapter, you will

- Get to know the application we will be building together
- Learn how to set up the environment to build the application

CHAPTER 2

Building Our First Application

Abstract

In this chapter, you will learn how to set up your development environment across all of the required platforms. You will then use that environment to create, build, and run your very first .NET MAUI application. Finally, you will take a look at the application you will build as you progress through this book.

It is worth noting that before setting up your environment, you cannot support all platforms from a single environment. On Windows, you can build for Windows and Android; you can also build for iOS, but you need to connect Visual Studio to a Mac. On macOS, you can build for iOS, Android, and macOS but not Windows. You can also develop on Linux and build for Android; however, we won't be covering how to configure a Linux development environment in this book. Table 2-1 shows the compatibility between development environment operating systems and which platforms can be built for.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_2

***Table 2-1.** The breakdown of which platform can be built on each operating system*

Environment	Can build for platform			
	Android	iOS	macOS	Windows
Windows	Yes	Yes*	No	Yes
macOS	Yes	Yes	Yes	No
Linux	Yes	No	No	No

**Through the use of a Mac connected to Visual Studio*

Setting Up Your Environment

Before you get into creating and building the application, you must make sure you have an environment set up.

macOS

There are several tools that you must install on macOS to allow support for building Mac Catalyst applications and to provide the ability to build iOS applications from a Windows environment.

This is required if you wish to develop on macOS or deploy to a Mac or iOS device (even from a Windows machine). If you are happy with only deploying to Windows or Android from a Windows machine, then you can skip this part or just read it for reference.

Xcode

Xcode is Apple's IDE for building applications for iOS and macOS. You don't need to use Xcode directly, but Visual Studio needs it in order to compile your iOS and macOS applications.

Thankfully this install is straightforward despite it being a rather large download.

1. Open the **App Store** application.
2. Enter *Xcode* into the **Search** box and press return.
3. Click **Get**. Figure 2-1 shows Xcode available on the Apple App Store.

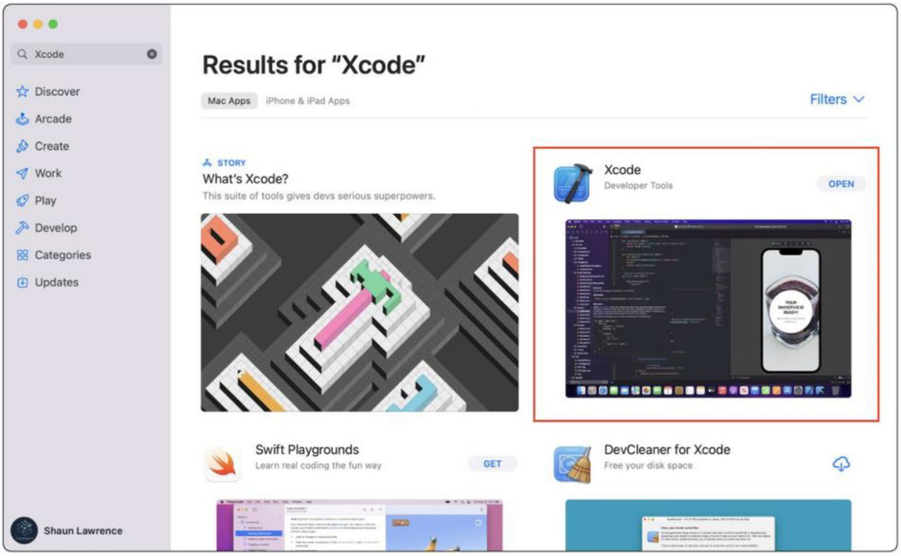

Figure 2-1. *Xcode on the App Store*

4. Once downloaded, open Xcode and wait for it to install the command-line tools. Note that this is usually required to be performed after each major update to Xcode, too.

I suggest using caution when applying updates to the whole suite of applications that you are installing today. Typically, when a new, big release of .NET MAUI comes out, it likely requires an update of Xcode. I personally like to keep these expected versions in sync so I recommend checking for the updates within Visual Studio first and verifying that it expects a new version of Xcode before proceeding to update that.

To aid the effort in maintaining the version of Xcode installed on your system, I would also thoroughly recommend the Xcodes tool which can be found at `https://github.com/XcodesOrg/XcodesApp`. This tool allows you to easily control which version is installed and even have multiple versions installed and select the default for use. This can be especially useful when wanting to investigate beta features while still working on production versions.

Remote Access

The final step to set up the macOS environment is to enable remote login so that Visual Studio (Windows) can communicate to the Mac to build and run iOS and macOS applications.

1. Open **System Settings** (macOS Ventura 13.0+) or System Preferences on older macOS versions.

2. Select **General** on the left-hand panel and then **Sharing**, as highlighted in Figure 2-2. This image shows the macOS System Settings dialog with the Sharing menu option highlighted.

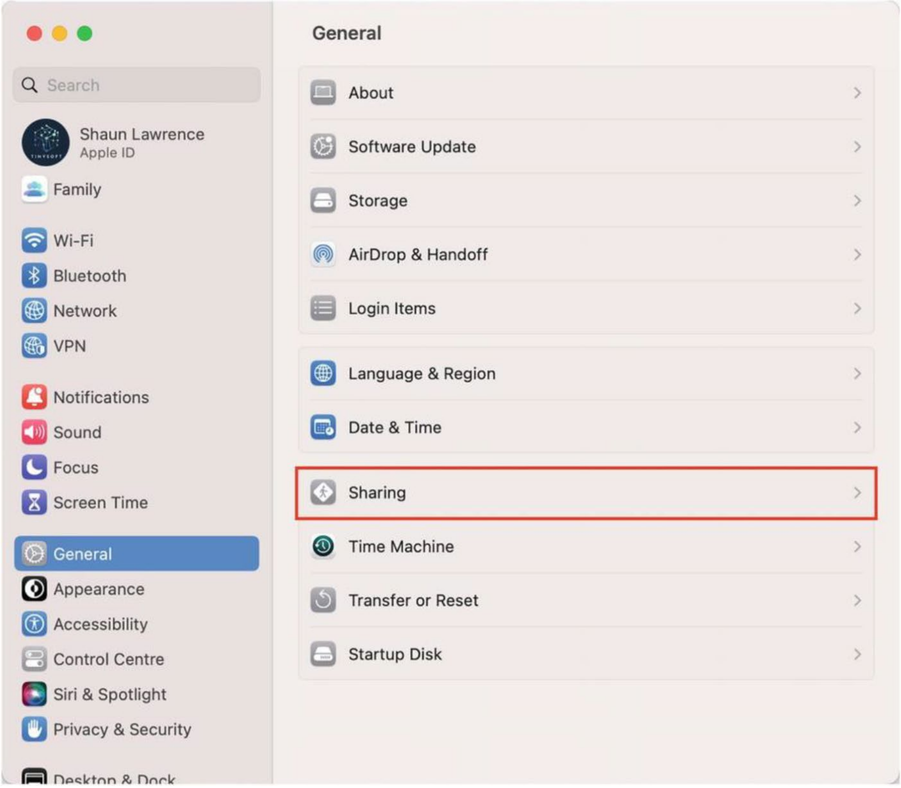

Figure 2-2. *macOS system settings*

3. Enable **Remote Login**. Figure 2-3 shows the Remote Login option enabled.

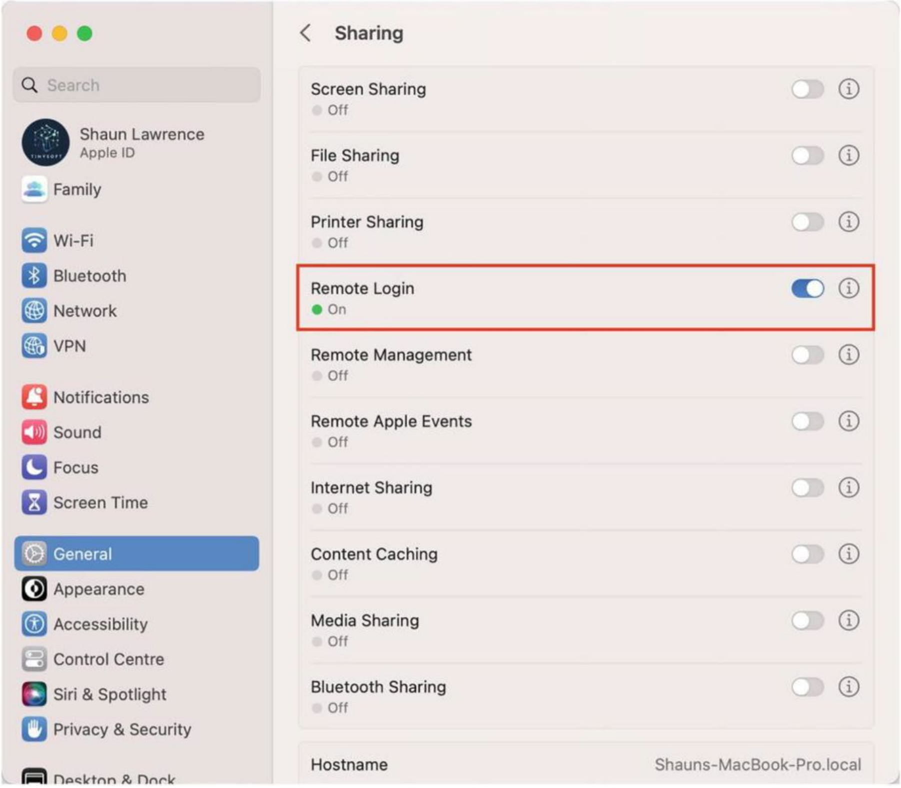

Figure 2-3. *macOS sharing options*

4. Add your user to the list of allowed users for Remote Login. My user is an Administrator so the Administrators user group enables remote login access for this user. Figure 2-4 shows the Remote Login editor to enable access for users on macOS.

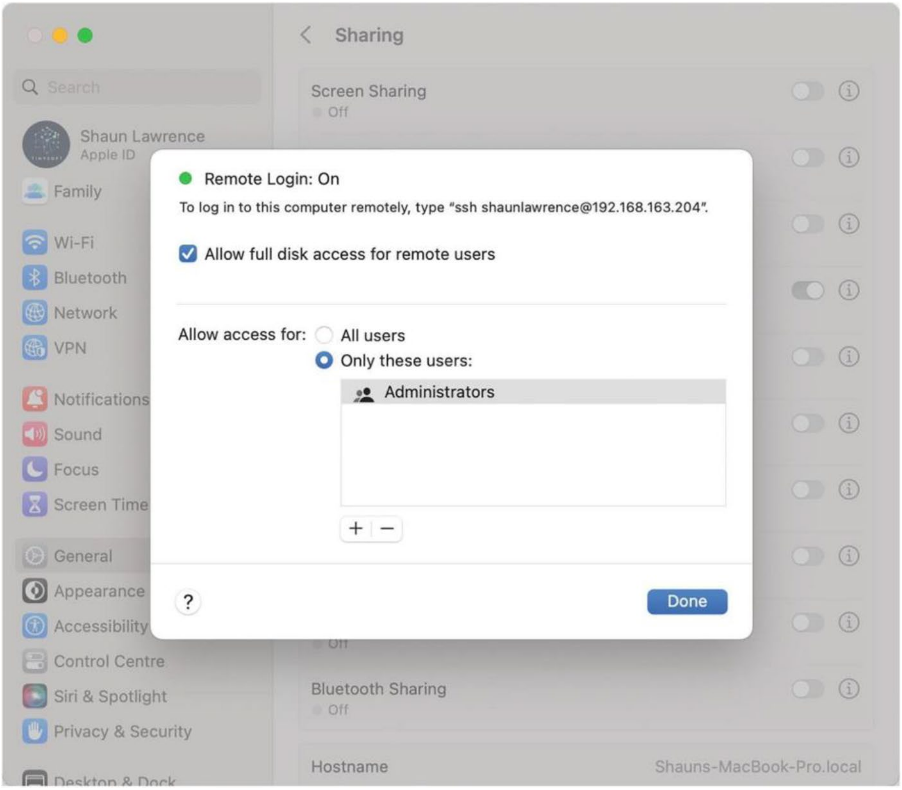

Figure 2-4. *macOS remote login options*

5. That's it! Your Mac should now be ready to use.

Windows

Visual Studio

First, you must install Visual Studio 2022. These steps will guide you through getting it ready to build .NET MAUI applications:

1. Download and install Visual Studio 2022. This can be accessed from `https://visualstudio.microsoft.com/downloads/`.

2. Run the installer and you will see the workload selection screen. Select the **Mobile development with .NET workload**. Figure 2-5 shows the Visual Studio Windows installer with the required .NET MAUI workloads checked.

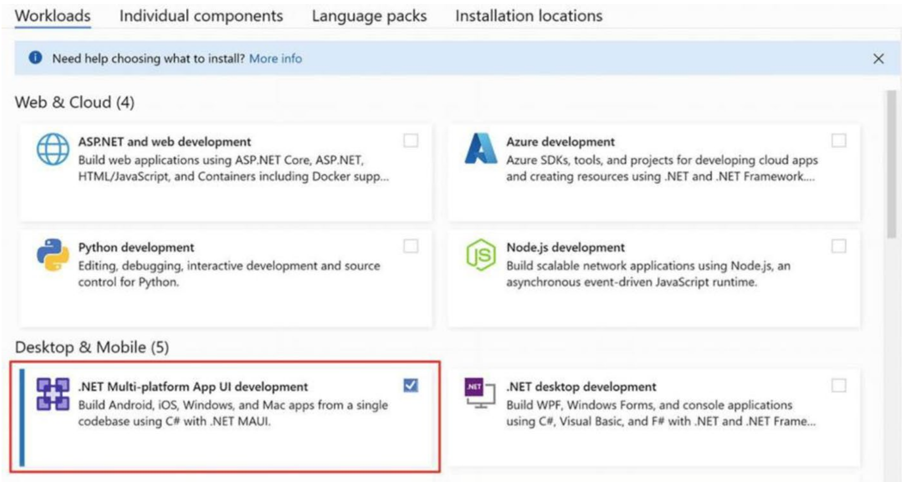

Figure 2-5. *Visual Studio Windows installation options*

Please refer to the Microsoft documentation page at `https://learn.microsoft.com/dotnet/maui/get-started/installation?tabs=vswin` if any of the installation options have changed.

Enable Developer Mode

In order to run your .NET MAUI application on your Windows machine through Visual Studio, you will first need to enable Developer Mode. This can be done by the following steps:

1. Open the Settings application.
2. Type *Developer* in the search bar.

3. Select Developer Settings.

4. Enable the Developer Mode switch (see Figure 2-6).

5. Read and accept the dialog that pops up.

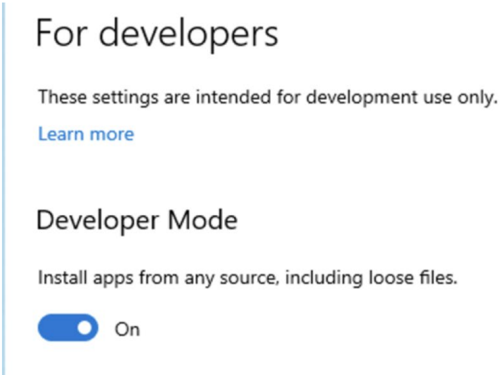

Figure 2-6. *The Developer Mode option in settings*

Visual Studio to macOS

The final item to configure in your Windows environment is to set up the connection between Visual Studio and your macOS so that iOS and macOS builds can be compiled.

1. Inside Visual Studio, select the Tools menu item.

2. Select iOS ➤ Pair to Mac.

3. Check and confirm the firewall access. Figure 2-7 shows the firewall request dialog that is presented when first running Visual Studio on Windows.

Figure 2-7. *Windows firewall request*

4. Select your Mac from the list.
5. Click **Connect**. Figure 2-8 shows the Pair to Mac dialog that allows you to connect your Visual Studio running on Windows to your macOS machine.

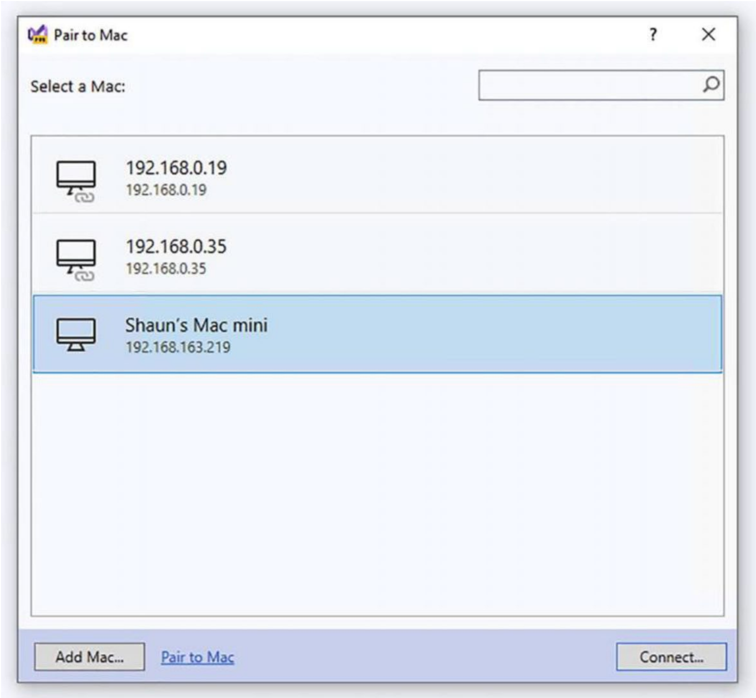

***Figure 2-8.** Pair to Mac screen*

6. Enter the username and password that you use to log into your Mac.

7. Wait for the tooling to connect and make sure that everything is configured on the Mac.

8. When you see the symbol shown in Figure 2-9, your setup is complete. Figure 2-9 shows the Pair to Mac dialog with the connected symbol against your macOS machine.

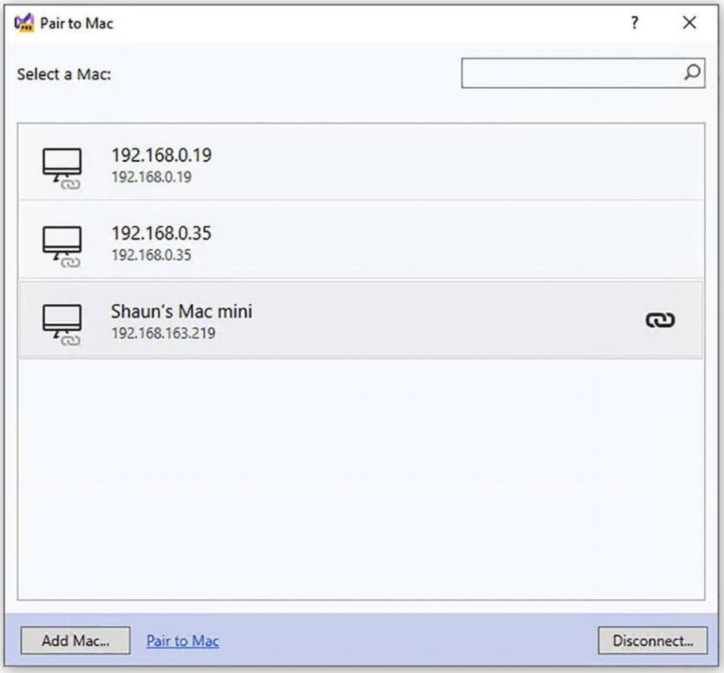

Figure 2-9. *Pair to Mac screen with confirmation*

9. Visual Studio should now connect automatically when you open a .NET MAUI solution. Figure 2-10 shows the Pair to Mac button in Visual Studio on Windows.

Figure 2-10. *Visual Studio toolbar with Pair to Mac buttons*

Troubleshooting Installation Issues

Given that there are several moving parts in the development ecosystem when building .NET MAUI applications, there is room for things to go wrong. In this section, I will go over a few common issues and how to check that things are correctly set up.

.NET MAUI Workload Is Missing

In order to check whether the .NET MAUI workload has been installed, you can check either in Visual Studio Installer or through the command line.

Visual Studio Installer

This currently only works on Windows, but you can follow these steps.

1. Open the **Start** menu.
2. Type in *Visual Studio Installer*.
3. Open the installer.
4. Select **Modify** on the Visual Studio 2022 installation.
5. View the workloads and check that the **Mobile development with .NET** workload is ticked.

Command Line

The command that we wish to run has the benefit of working on both Windows and macOS, but opening a command prompt or terminal session is different on each operating system. Let's take a look at each in turn.

macOS

1. Open the **Terminal** application.
2. Enter the following command and then press return:

   ```
   dotnet workload list
   ```

Windows

1. Open the **Command Prompt** application.
2. Enter the following command and then press return:

   ```
   dotnet workload list
   ```

Results for Both

Both operating systems and applications will report the same results:

```
Installed Workload Id Manifest Version Installation Source
------------------------------------------------------
maui                 9.0.X/9.0.X  SDK 9.0.X
```

You should verify that the results include **maui** and that they are of the expected version. For example, the current version is .NET MAUI 9.0 so we are looking for the Manifest and Version to start with 9.0. If you are working with a different major version, then confirm that it is installed.

If the version is not installed, you can then enter the following command in your Command Prompt or Terminal application and press return:

```
dotnet workload install maui --sdk-version=9.0.X
```

where the X above is the version you wish to install.

Creating Your First Application

You will be using the user interface in order to create your application, build, and run it. I will also be including the dotnet command-line commands because I find they can be quite helpful when building and debugging.

Creating in Visual Studio

1. Launch Visual Studio 2022. In the window that opens, select the ***Create a new project*** option. Figure 2-11 shows the initial starting screen in Visual Studio running on Windows with the ***Create a new project*** option highlighted.

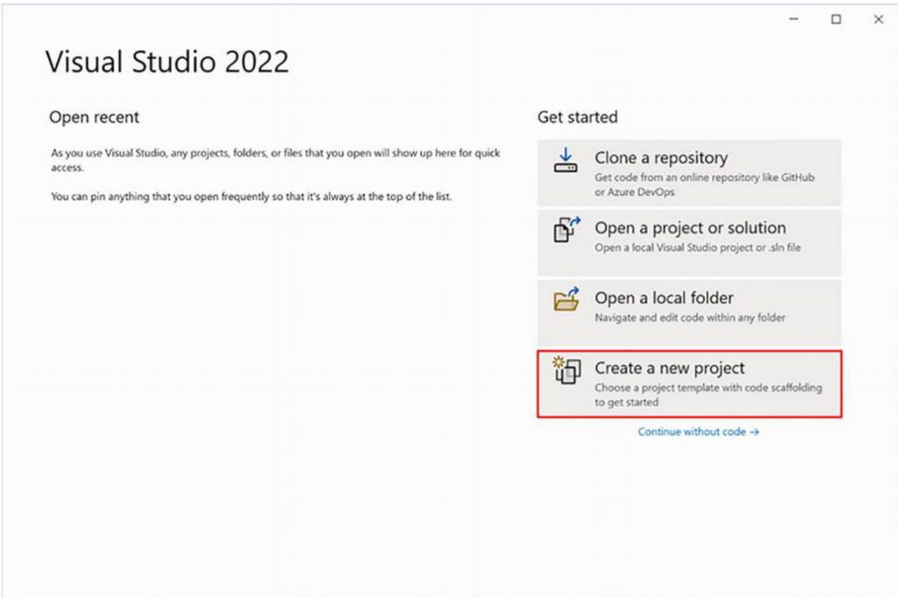

Figure 2-11. *Creating a project in Visual Studio*

2. In the window that follows, type *.NET MAUI* in the Search for templates box. Then select the **.NET MAUI App** option and click **Next**. Figure 2-12 shows the project creation screen with the .NET MAUI App project selected.

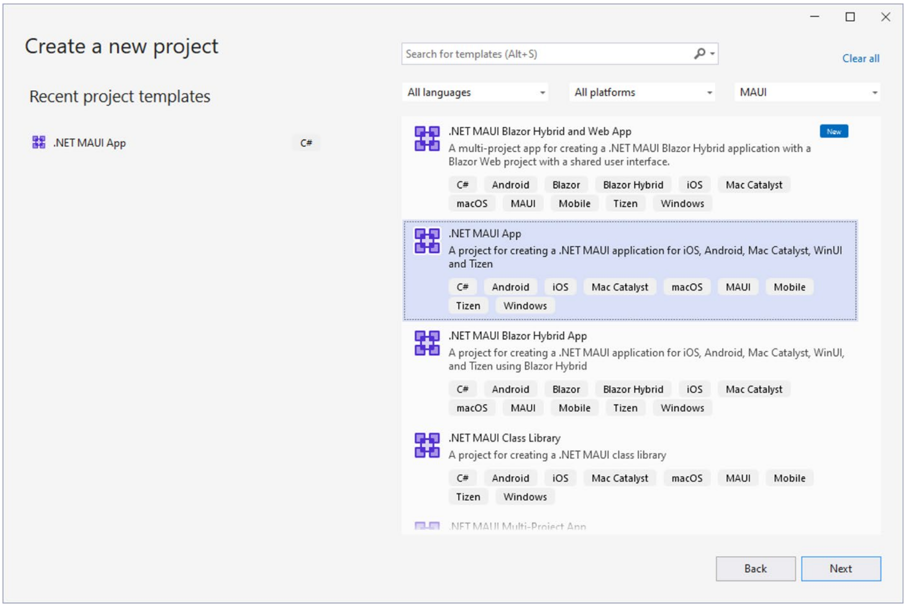

***Figure 2-12.** Selecting a .NET MAUI App project type*

3. In the next window, enter a name for your project. I chose WidgetBoard. Choose a location if you would like to store it somewhere different from the default location, and click **Create**. Figure 2-13 shows the Configure your new project screen in Visual Studio.

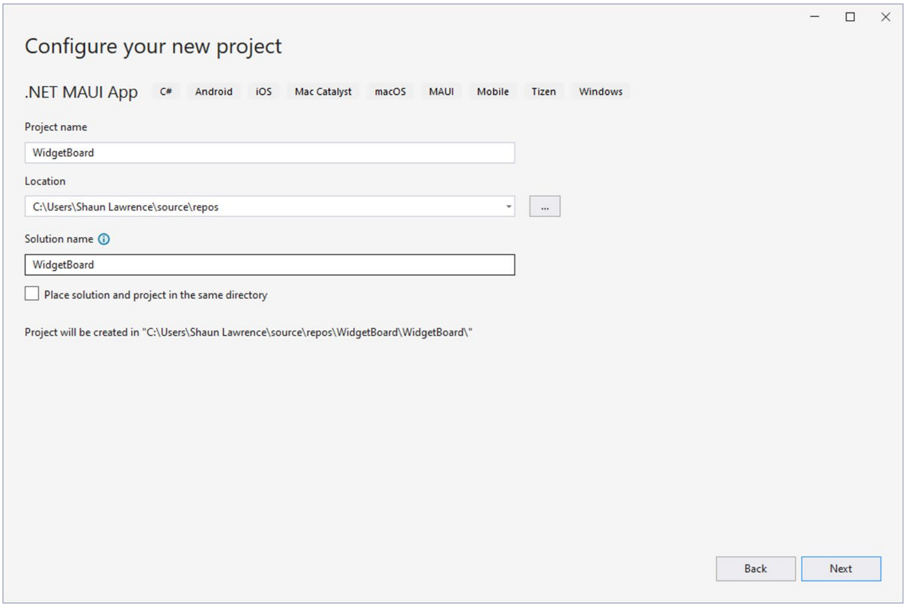

Figure 2-13. *The Configure your new project dialog*

Please bear in mind that Windows has a limitation on the length of the location path. If the path is longer than 255 characters, then strange behavior will follow. Visual Studio will fail to build perfectly valid code and so on. This can be rectified by disabling the path limit (`https://learn.microsoft.com/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later`).

4. Select the version of .NET you wish to use. At the time of writing this book, .NET 9.0 is the current version so I am using this version. Figure 2-14 shows the Additional information dialog where you can choose the .NET Framework version for your application.

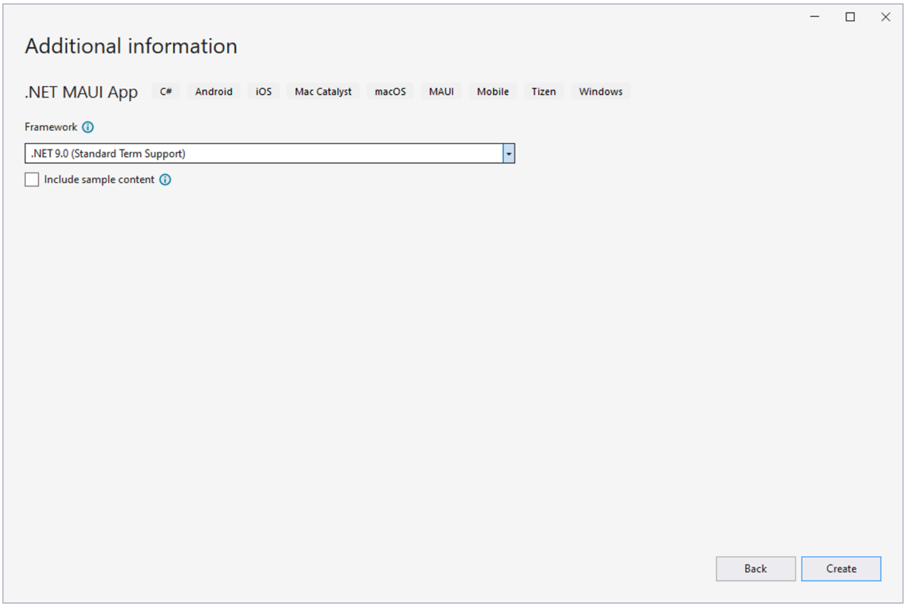

Figure 2-14. *The .NET Framework selection dialog*

5. Wait for the project to be created and any background restore and build tasks to be completed.

Now admire the very first .NET MAUI application that we have created together. Note that we didn't tick the *Include sample content* check box on the last page; this is because the sample content doesn't fit our scenario and we would end up deleting most of it. I would recommend creating a project with this option ticked to gain an understanding of what they provide as an example of how to build a good-looking application.

Creating in the Command Line

While the command line might feel more complicated, at times there are actually fewer steps required than when using Visual Studio.

1. Open a Terminal/command-line session.
   - On macOS, open the **Terminal** application.
   - On Windows, open the **Command Prompt** application.
2. cd to the location you want to create your application:

   ```
   cd c:\work\
   ```
3. Create the application, giving the project a name:

   ```
   dotnet new maui --name WidgetBoard
   ```
4. cd to the new folder, `WidgetBoard`:

   ```
   cd WidgetBoard
   ```
5. Pull in all dependencies for the application:

   ```
   dotnet restore
   ```

You now have a .NET MAUI application. Let's proceed to learning how to build and ultimately run it.

Building and Running Your First Application

Now that you have your project created, let's go ahead and build and run it in order to get familiar with the tooling. If this is the first time that you are building and running a .NET MAUI application for Android, you will likely see a prompt helping you to create an Android emulator. Please proceed through this before you try to run the application on Android.

The introduction of the single project approach for .NET MAUI applications may bend your way of thinking when it comes to building applications. In the past, a solution containing .NET projects would typically have a single start-up project, but these projects would have a single output. Now that a single project actually has multiple outputs, you need to learn how to configure that for your builds. In fact, this is done by clicking the down arrow, which can be seen in Figure 2-15.

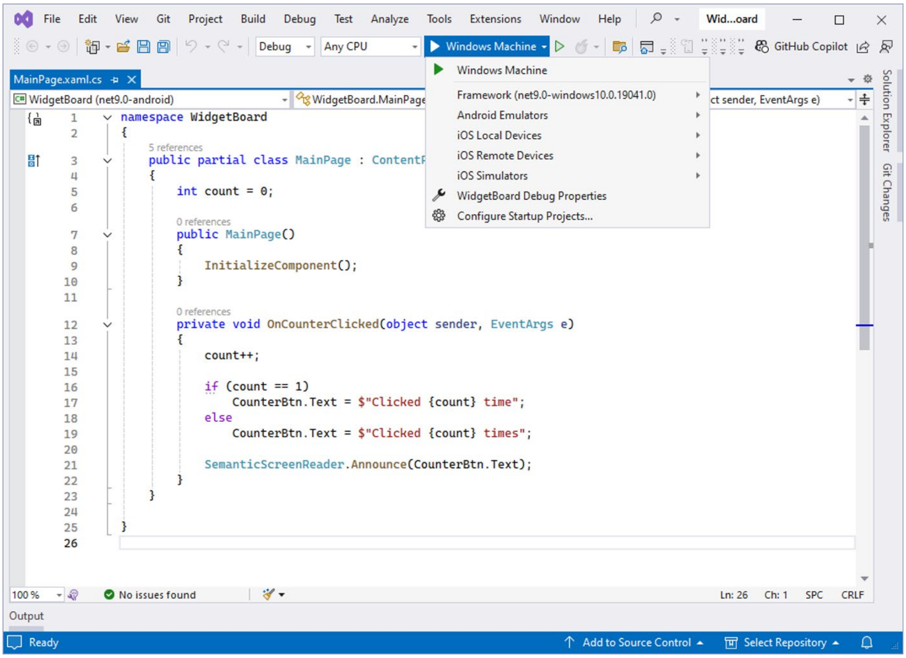

***Figure 2-15.** Build target selection drop-down in Visual Studio*

You may also notice the drop-down in the above image that currently says **WidgetBoard (net9.0-android)**. This allows you to show in the visible file what applies to that specific target, but it does not affect what you are currently compiling. Figure 2-16 shows this a little clearer.

1. This is where you set the current target to compile for and run.

2. This is highlighting in the code file what will compile for the target chosen in the drop-down. Notice here that you are compiling for Windows but showing what would compile for Android.

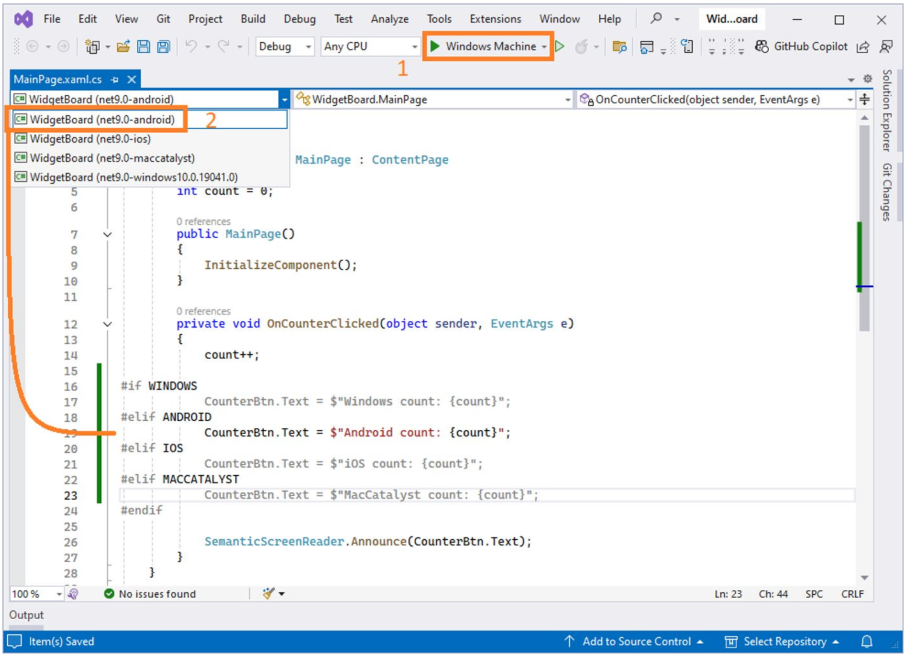

***Figure 2-16.** The differences between what target is being compiled and what target is being shown in the current editor*

Figure 2-16 highlights items 1 and 2 from the above list to highlight what is compiled vs. what is targeted in Visual Studio.

Getting to Know Your Application

Together we will be building an application from the very initial stages through to deploying it to stores for public consumption. Given that the application will play such a pivotal role in this book, I want to introduce you to the concept first.

I want to try something a little bit different from the normal types of apps that are built as part of a book or course. Something that requires a fair amount of functionality that a lot of real-world applications also need. Something that can help to make use of potentially older hardware so we can give them a new lease on life.

WidgetBoard

The application that we will be building together will allow users to turn old tablets or computers into their own unique digital board. Figure 2-17 shows a sketch of how it could look once a user has configured it.

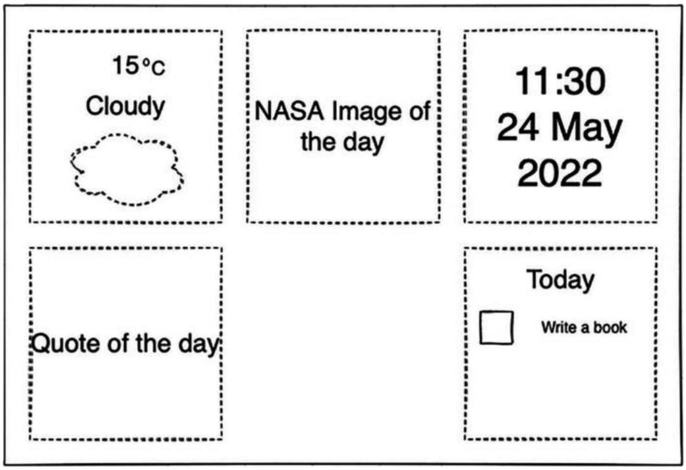

Figure 2-17. *Sketch prototype of the application we will be building*

We will build "widgets" that can be positioned on the screen. These widgets will range from showing the current time to pulling weather information from a web API to displaying images from your library.
The user will also be able to customize the color, among other options, and ultimately save these changes so that they will be remembered when the user next opens the application.

I am planning for this to provide a digital calendar/photo frame for our home. I would love to hear or see what you are able to build.

Summary

In this chapter, you have

- Set up your development environment so that you are capable of creating, building, and ultimately running/deploying the application
- Created, built, and run your very first .NET MAUI application
- Met the application that we will be building together

In the next chapter, you will

- Dissect the application you just created
- Gain an understanding of the key components of a .NET MAUI application
- Learn about the life cycle of a .NET MAUI application

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch02`.

CHAPTER 3

The Fundamentals of .NET MAUI

Abstract

In this chapter, you will dissect the project you created in Chapter 2 and dive into the details of each key area. The focus is to provide a good overview of what a .NET MAUI single project looks like, where each of the key components are located, and some common ways of enhancing them.

Project Structure

.NET MAUI provides support for multiple platforms from within a single project. The focus is to allow us as developers to share as much code and as many resources as possible.

You will likely hear the term *single project* a lot during your time working with .NET MAUI. It is a concept that is relatively new to the .NET world, introduced as part of .NET MAUI. Its key feature is that you can build applications for multiple different targets from, you guessed it, a single project. If you have ever built .NET applications that aim to share code, you will have noticed that each application you wanted to build and

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_3

deploy required its own project. The same was true with Xamarin.Forms in that you would have at least one project with your common code and then one project per platform. The single project now houses both the shared code and the platform-specific bits of code.

Figure 3-1 shows a comparison between the old separate project approach in Xamarin.Forms and the new .NET MAUI project format. The squares represent a project file.

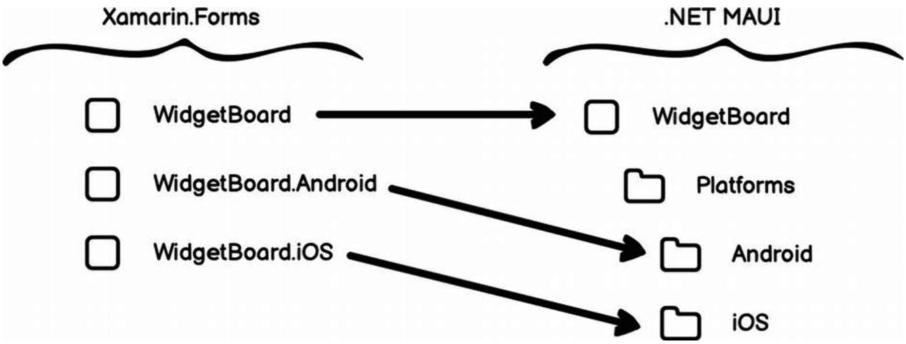

Figure 3-1. *Comparison of Xamarin.Forms projects to a .NET MAUI project*

Let's inspect the project you created in Chapter 2 so that you can start to get an understanding of how .NET MAUI supports the multiple platforms and how they relate to shared code.

The new project has the following structure:

- `Platforms/`: This folder contains all the platform-specific code. Inside this folder is a set of folders, each with a name that relates to the platform that it supports. Thus, `Platforms/Android` supports the Android platform, `Platforms/iOS` supports the iOS platform, and so on.

- `Resources/`: This folder is where you store all your resources for the application. A resource is typically anything you wish to embed in the application that isn't strictly code, such as an image, a font file, or even an audio or video file.

- `MauiProgram.cs`: This class is where you initialize your .NET MAUI application. It makes use of the Generic Host Builder, which is the Microsoft approach to encapsulating the requirements of an application. These requirements include but are not limited to dependency injection, logging, and configuration.

- `App.xaml.cs`: This is the main entry point to the cross-platform application. Note this line of code from the `MauiProgram.cs` file includes our `App` class:

  ```
  builder.UseMauiApp<App>();
  ```

- `App.xaml`: This file includes common UI resources that can be used throughout the application. I will cover these types of resources in much more detail in Chapters 5 and 8.

- `MainPage.xaml` and `MainPage.xaml.cs`: These two files combine to make up your application's first page.

- `AppShell.xaml` and `AppShell.xaml.cs`: These two files enable you to define how your application will be laid out through the use of the .NET MAUI concept called *Shell*. I will cover Shell extensively in Chapter 5.

Note that wherever you see a `.xaml` file, there will typically be an associated `.xaml.cs` file. This is due to limitations in what XAML can provide; it requires an associated C# file to cover the parts that XAML does not support. I will cover XAML much more extensively in Chapter 5.

It is also worth noting that you do not have to write any XAML. Sure, .NET MAUI and its predecessor, Xamarin.Forms, have a deep connection to XAML, but because the XAML is ultimately compiled down to C#, anything that is possible to create in XAML is also possible in C#. The next chapter (Chapter 4) will take you through the different possibilities for architecting your applications.

/Platforms/ Folder

I mentioned that the platform-specific code lives in the `Platforms` folder. While cross-platform applications provide a nice abstraction from the platforms we wish to support, I still believe it is extremely valuable to know how these platforms behave. Let's dive in and look at each of the platform folders to understand what is happening.

Android

Inside the Android platform folder, you will see the following files:

- `Resources/values/colors.xml`: This contains color information used for the Android platform. If you wish to change some of the colors used within your application, you will need to update this file.
- `MainApplication.cs`: This is the main entry point for the Android platform. Initially you should note that it does very little. The bit it does is rather important, though; it is responsible for creating the `MauiApp` using the `MauiProgram` class. This is the bridge between the Android application and your cross-platform .NET MAUI code.

- `MainActivity.cs`: An activity in Android development is a type of app component that provides a user interface. The `MainActivity` starts when your app is loaded. This is typically done by tapping the app icon; however, it can also be triggered by a notification or other source.
- `AndroidManifest.xml`: This file is extremely important. It is how you define the components that make up your application, any permissions it requires, the application version information, the minimum and target SDK versions, and any hardware or software features that it requires.

iOS

Inside the iOS platform folder, you will see the following files:

- `AppDelegate.cs`: This class allows you to respond to all platform-specific parts of the application lifecycle.
- `Info.plist`: This file contains configuration about the application. It is like the `AndroidManifest.xml` file discussed in the "Android" section. You can change the application's version and include reasons why your application requires permission to use certain features.
- `Program.cs`: This is the main entry point.

MacCatalyst

Inside the MacCatalyst platform folder, you will see the following files. It is worth noting that this section is nearly identical to the previous iOS section. It's been kept separate to provide an easy reference to what the platform folder consists of for MacCatalyst.

- `AppDelegate.cs`: This class allows you to respond to all platform-specific parts of the application lifecycle.
- `Entitlements.plist:` This file contains a key-value pair list of capabilities that your macOS application requires.
- `Info.plist`: This file contains configuration about the application. It is like the `AndroidManifest.xml` file discussed in the "Android" section; you can change the application version and include reasons why your application requires permission to use certain features.
- `Program.cs`: This is the main entry point.

Tizen

Inside the Tizen platform folder, you will see the following files:

- `Main.cs`: This is the main entry point for your Tizen application.
- `tizen-manifest.xml`: This file is very similar to the `AndroidManifest.xml` file. It is how you define the components that make up your application, any permissions it requires, the application version information, the Tizen API version, and any hardware or software features it requires.

Windows

Inside the Windows platform folder, you will see the following files:

- `app.manifest`: The package manifest is an XML document that contains the info the system needs to deploy, display, or update a Windows app. This info includes package identity, package dependencies, required capabilities, visual elements, and extensibility points. Every app package must include one package manifest.
- `Package.appxmanifest`: An application manifest is an XML file that describes and identifies the shared and private side-by-side assemblies that an application should bind to at runtime. They should be the same assembly versions that were used to test the application. Application manifests may also describe metadata for files that are private to the application.

Summary

Phew! That felt like a lot to take in! I think I need to take a tea break! Don't worry, though; while this gives an overview of what each of the files is responsible for, you will be modifying most of them throughout this book with some practical examples, so if there are any points that aren't clear, or you feel you will need to revisit them, you certainly will be.

/Resources/ Folder

The `Resources` folder is where you store anything you want to include in your application that is not strictly code. Let's look through each of the subfolders and key types of resource.

AppIcon

This aptly named folder is responsible for housing the icon image files used to generate our application's icon. The default project that is created provides us with two images in this folder. In Chapter 5, you will learn how to replace the defaults and how the app icons are structured. This type of resource is called a `MauiIcon`.

Fonts

.NET MAUI allows you to embed your own custom fonts. This is especially handy if you are building an app for a specific brand, or you want to make sure that you render the same font on each platform. You can embed either True Type Fonts (`.ttf` files) or Open Type Fonts (`.otf` files).

A word of warning around fonts. I strongly recommend that you check the licensing rules around fonts before including them in your application. While there are sites that make it possible to download fonts freely, a very large percentage of those fonts usually require paying to use them.

There are two parts to embedding a font so that it can be used within your application.

1. The font file should be placed in this folder (`Resources/Fonts`).

   By default, the font will be automatically included as a font file based on the following line that can be found inside the project file (`WidgetBoard.csproj`):

   ```
   <MauiFont Include="Resources\Fonts\*" />
   ```

   What the above line does is set the **build action** of the file you just included to be of type `MauiFont`. If you want to perform this manually, you can right-click the file inside Visual Studio, click **Properties**, and inside the Properties panel, set the **Build Action** to **MauiFont**.

2. Configure the font.

   When bootstrapping your application, you need to specify which fonts you wish to load. This is performed with the following lines inside your `MauiProgram.cs` file:

```
.ConfigureFonts(fonts =>
{
    fonts.AddFont("Lobster-Regular.ttf", "Lobster");
});
```

In the above example, you add the font file `Lobster-Regular.ttf` to the collection of fonts and give it an alias of **Lobster**. This means you can just use the name of Lobster when referring to the file in your application.

Images

Practically every application you build will include some images. Each platform that you wish to support has its own rules on the image sizes that you need to supply to make the image render as sharp and clear on the many devices they run. Take iOS, for example. In order to supply a 24×24 pixel image in your app, you must provide three different image sizes: 24×24, 48×48, and 72×72. This is due to the different DPIs for the devices Apple builds. Android devices follow a similar pattern, but the DPIs are not the same. This is similar for Windows.

Figure 3-2 shows an example image that would be rendered at 24×24 pixels. Note that while Windows shows the three sizes, this is just based on recommendations for trying to cover the most common settings. In truth, Windows devices can have their DPIs vary much more. Figure 3-2 shows the required image sizes needed for all supported platforms in order to render a 24×24 pixel image.

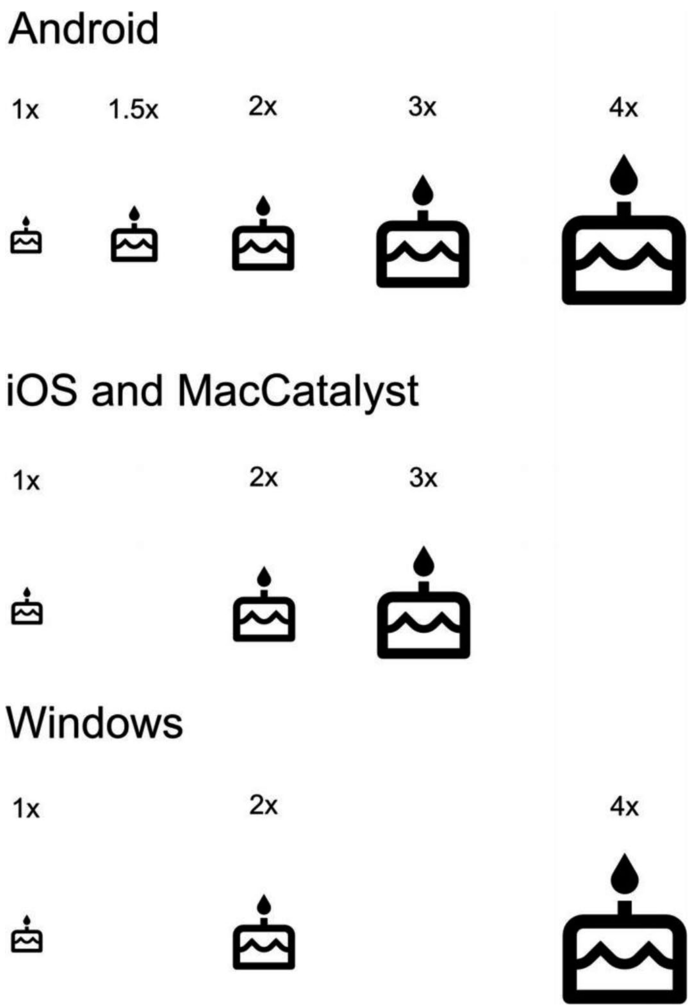

Figure 3-2. *Required image sizes across the various platforms*

You can see from the figure above that it can become painful very quickly if you have a lot of images in your application each requiring at least five different sizes to be maintained. Thankfully .NET MAUI gives us the ability to provide a single Scalable Vector Graphics (SVG) image, and it will generate the required images for all the platforms when the application is compiled. I cannot tell you how happy all of us Xamarin. Forms old timers are at this new piece of functionality!

As it currently stands, if the SVG image is of the correct original size, you can simply drop the image into the `/Resources/Images/` folder and it will just begin to work in your application. In a similar way to how the fonts are automatically picked up, you can see how the images are also handled by looking inside your project file and observing the line `<MauiImage Include="Resources\Images\*" />`.

.NET MAUI doesn't render SVGs directly but generates PNG images from the SVGs at compile time. This means that when you are referring to the image you wish, it needs to have the `.png` extension. For example, when embedding an image called `image.svg`, in code, you refer to it as `image.png`.

If the contents of the SVG are not of the desired size, then you can add some configuration to tell the tooling what size the image should be. For this, the image **should not** be added to the `/Resources/Images/` folder as the tooling will end up generating duplicates and there is no telling which one will win. Instead, you can simply add the image to the `/Resources/` folder and then add the following line to your project file:

```
<MauiImage Include="Resources\image.svg" BaseSize="24,24" />
```

The above code will treat the contents of the `image.svg` file as being 24×24 pixels and then scale for each platform based on that size.

Raw

The next type of resource to embed is raw files. This essentially means that what is embedded can be loaded at runtime. A typical example of this is to provide some data to preload into the application when first starting. This type of resource is called a `MauiAsset`.

Splash

This folder is created by default to show how a splash screen can be added to a .NET MAUI application. In Chapter 5, you will learn how to customize the defaults and provide your own splash screen along with the many ways a splash screen can be customized. This type of resource is called a `MauiSplashScreen`.

Styles

The Styles folder is where developers are encouraged to create style-related resources; these could be control styles, color palettes, or even CSS styles. We will cover these items throughout the book with the main focus being in Chapter 5. There isn't a single type of resource for the contents of this folder but the two defaults created for us; Colors.xaml and Styles.xaml are of type MauiXaml, and these will be the most common type of resources that you will create here.

This concludes the section on the */Resources/* folder. Let's proceed to covering where an application begins its life.

Where to Begin?

.NET MAUI applications have a single main entry point that is common across all platforms. This provides us with a way to centralize much of the initialization process for our applications and therefore only write it once.

You will have noticed that in each of the platform-specific main entry points covered in the previous section, they all call `MauiProgram.CreateMauiApp();`. This is the main entry point into your .NET MAUI and shared application.

The `CreateMauiApp` method allows you to bootstrap your application. Bootstrapping refers to a self-starting process that is supposed to continue or grow without external input (Wikipedia quote). This means that your implementation in this method is responsible for configuring the application from setting up logging, general application configuration, and registering implementations to be handled with dependency injection. This is one of the big improvements in .NET MAUI over Xamarin.Forms. This is done through the Generic Host Builder.

Generic Host Builder

I mentioned back in Chapter 1 that one of the benefits that comes with the evolution to .NET MAUI is powerful dependency injection. The Generic Host Builder is tried and tested through other .NET frameworks such as ASP.NET Core, and it has thankfully become available to all application types now.

Before we jump into how the Generic Host Builder works, let's look at what exactly dependency injection is and why you should use it.

What Is Dependency Injection?

Dependency injection (DI) is a software design pattern aimed at reducing hard-coded dependencies in a software application. A dependency is an object that another object depends on. This hard-coded dependency approach is referred to as being tightly coupled. Let's work through an example to show how and why it's named so and how you can remove the need for the hard-coded dependencies, thus making your design loosely coupled.

So, my wife is a fantastic baker. She bakes these beautiful, delicious cakes, and this is the main reason I have gained so much weight recently. I am going to use the process of her baking a cake to show this concept of dependencies.

```
public class Baker
{
    public Cake Bake()
    {
    }
}
```

The above code looks relatively straightforward, right? She bakes a cake. Now let's consider how she might go about making the cake. She needs a way of sourcing the ingredients, weighing them, mixing them, and finally baking them. We end up with something like

```
public class Baker
{
    private readonly WeighingScale weighingScale = new
    WeighingScale();
    private readonly Oven oven = new Oven();
    private readonly MixingMachine mixingMachine = new
    MixingMachine();
    private readonly IngredientsProvider ingredientsProvider =
    new IngredientsProvider();
    public Cake Bake()
    {
        Ingredient ingredient = ingredientsProvider.Provide();
        weighingScale.Weigh(ingredient);
    }
}
```

We can see that for the Baker to do their job, they need to know about all these different pieces of equipment. Now imagine that the WeighingScale breaks, and a replacement is provided. The Baker will still need to weigh the ingredients but won't care how that weighing is performed. Imagine that the new WeighingScale is digital and now requires batteries. There are a few reasons why we want to move away from having hard-coded dependencies as in our Baker example.

- If we did replace the WeighingScale with a different implementation, we would have to modify the Baker class.
- If the WeighingScale has dependencies (e.g., batteries in our new digital scale), they must also be configured in the Baker class.
- This becomes more difficult to unit test because the Baker is creating dependencies and therefore a unit test would result in having to test far more than a unit test is designed to.

Dependency injection can help us to address the above issues by allowing us to achieve *Inversion of Control* (IoC). Inversion of Control essentially means that we are inverting the knowledge of the dependency from the Baker knowing about a WeighingScale to them knowing about something that can weigh ingredients but not an actual implementation. This is done through the introduction of an interface which we will call IWeighingScale.

```
public class Baker
{
    private readonly IWeighingScale weighingScale;
    private readonly Oven oven = new Oven();
    private readonly MixingMachine mixingMachine = new
    MixingMachine();
```

```
    private readonly IngredientsProvider ingredientsProvider =
    new IngredientsProvider();
    public Baker(
        IWeighingScale weighingScale)
    {
        this.weighingScale = weighingScale;
    }
    public Cake Bake()
    {
        Ingredient ingredient = ingredientsProvider.Provide();
        this.weighingScale.Weigh(ingredient);
    }
}
```

Now our `Baker` knows about an interface for something that can weigh their ingredients but not the actual thing that does the weighing. This means that in the scenario where the weighing scale breaks and a new one is supplied, there is no change to the `Baker` class in order to handle this new scale. Instead, it is registered as part of the application startup or bootstrapping process. Of course, we could and should follow the same approach for our other dependencies.

One additional concept I have introduced here is the use of *constructor injection*. Constructor injection is the process of providing the registered dependencies when creating an instance of our `Baker`. So, when our `Baker` is created, it is passed an instance of `WeighingScale`.

If you have a background with Xamarin.Forms, you will have come across the `DependencyService`. This provided a mechanism for managing dependency injection within an application; however, it received criticism in the past for not supporting constructor injection. This doesn't mean it wasn't possible to achieve constructor injection in Xamarin.Forms applications, but it required the use of a third-party package and there are a lot of great packages out there! Now it is all baked into .NET MAUI.

Registering Dependencies

In the previous section, I discussed how to minimize concrete dependencies in your code base. Now let's look through how to configure those dependencies so that the dependents are given the correct implementations.

Implementations that you register in the generic host builder are referred to as services, and the work of providing the implementations out to dependents is referred to as the `ServiceProvider`. You can register your services using the following.

AddSingleton

A singleton registration means that there will only ever be **one** instance of the object. So, based on the example of our `Baker` needing to use an `IWeighingScale`, we register it as follows:

```
builder.Services.AddSingleton<IWeighingScale, WeighingScale>();
```

Then every time that an `IWeighingScale` is resolved, we will be provided with the **same** instance. This suits the weighing scale example because we use the same one throughout our baking process.

It is extremely unlikely that you will ever need to register a view model as a singleton. Doing so can introduce bits of behavior that you are most likely not expecting on top of the fact that you can run the risk of leaking memory.

AddTransient

A transient registration is the opposite of a singleton. Every time an implementation is resolved, a **new** instance is created and provided. So based on the example of our `Baker` needing to use an `IWeighingScale`, we register it as follows:

```
builder.Services.AddTransient<IWeighingScale, WeighingScale>();
```

As mentioned, every time an `IWeighingScale` is resolved, we will be provided with a **new** instance. A better example here might be the greaseproof paper that lines the cake tins. They are used once and thrown away.

AddScoped

A scoped registration is somewhere in the middle of a singleton and transient. A single instance will be provided for a "scope," and then when a new scope is created, a new instance will be provided for the life of that scope.

```
builder.Services.AddScoped<IWeighingScale, WeighingScale>();
```

This type of registration feels much better suited to a web application where requests come in and a scope will represent a single request. In the mobile and desktop world, your application typically has a single state and therefore is less likely to need scoped registrations. Currently .NET MAUI does not provide any automatic creations of scopes, but you have the power to create your own using the `IServiceScopeFactory` interface and ultimately its implementation.

This concludes the section on the Generic Host Builder and dependency injection. Let's proceed onto learning about the lifecycle of an application.

Application Lifecycle

Sadly, no two platforms provide the same set of behaviors or lifecycle events such as when an application is started, backgrounded, or closed. This is where cross-platform frameworks provide us with a solid set of encapsulated events to cover most scenarios. There are four main application states in a .NET MAUI application.

Application States

These are the application states:

- Not running: This means that the application has not been started and is not loaded into memory. This is typically when the application has been installed, the device has been powered on, the application was closed by the user, or the operating system has terminated the application to free up some resources.
- Running: This means that the application is visible and is focused.
- Deactivated: This means that the application is no longer focused but may still be visible. On mobile, this could mean that the operating system is showing a permission request alert (e.g., an application asking for permission to use the camera) or similar.
- Stopped: This means that the application is no longer visible.

You can now see how a .NET MAUI application moves between the above four states and the events that are triggered to an application. Figure 3-3 shows the possible states that a .NET MAUI application can take during its lifetime and how it transitions between those states.

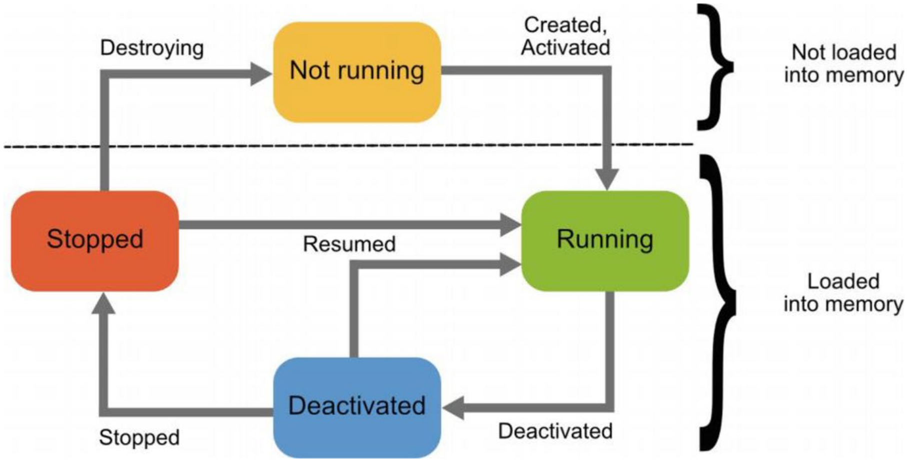

***Figure 3-3.** Application state lifecycle chart*

Before we dive into the details of each of the events that are fired between the state transitions, I need to give you some background on how they can be accessed and why. In order to access these events, you must access the `Window` class. It certainly isn't a common concept to have a window in a mobile application, but you must appreciate that you are dealing with a cross-platform framework and therefore an approach that fits desktop as well as mobile. I see it as follows: a mobile application is a single window application, and a desktop is likely to be multi-window.

Lifecycle Events

Now on to the events that move an application between states. These are the annotations on the arrows from Figure 3-3:

- `Created`: This event is raised after the platform window has been created. Note that the window may not be visible yet.

- `Activated`: This event is raised when the window is the focused window.

- `Deactivated`: This event is raised when the window is no longer the focused window. Note that the window may still be visible.
- `Stopped`: This event is raised when the window is no longer visible. The application may resume from this state but it is not guaranteed, so it is recommended that you cancel any long-running processes or anything that may consume resources on the device. Mobile operating systems are much stricter on what can happen in the background.
- `Resumed`: This event is raised when an application resumes from the `Stopped` state. It is recommended to prepare your application for full use again (e.g., subscribe to events or messages, refresh any visible content).
- `Destroying`: This event is raised when the platform window is being destroyed and removed from memory. It is recommended that you unsubscribe from events or messages.

Handling Lifecycle Events

By default, a .NET MAUI application won't give you access to the lifecycle events; this is something you must opt in for. In order to opt in, you must modify your `App` class.

Open Visual Studio. You need to add a new class to your project and call it `StateAwareWindow`. Your new class will need to be modified so it looks as follows:

```
public class StateAwareWindow: Window
{
    public StateAwareWindow() : base()
    {
    }
    public StateAwareWindow(Page page) : base(page)
    {
    }
    protected override void OnCreated()
    {
        // Initialise our application
    }
}
```

Inside of your application, you can override all methods that will be executed when the specific event occurs. Each override method follows the naming of the events, as described previously, with a prefix of `On`. Therefore, to handle the `Activated` event, you override the `OnActivated` method.

The final step is to make use of the new class, so inside your `App.xaml.cs` file, add the following:

```
protected override Window CreateWindow(IActivationState
activationState)
{
    return new StateAwareWindow(MainPage);
}
```

This will create a new instance of `StateAwareWindow` and pass it a reference to the application's `MainPage`. If you do not pass in a reference to a `Page` to the `Window` implementation, you will experience exceptions being thrown.

Cross-Platform Mappings to Platform Lifecycle Events

I strongly believe that despite the fact that .NET MAUI provides us with these unified events, you should understand how they map to the underlying platforms. If you understand what is being called on the platform-specific side, it can really help to diagnose things when they go wrong or perhaps point you in the direction of a better approach for your scenarios.

Let's break down how the .NET MAUI lifecycle events map to the platform-specific events and then show off the bits that are not mapped if you ever need to use them. See Table 3-1.

Table 3-1. *Cross-platform lifecycle events mapped to the platform-specific events*

Event	Android	iOS/Mac Catalyst	Windows
`Created`	`OnPostCreate`	`FinishedLaunching`	`Created`
`Activated`	`OnResume`	`OnActivated`	`Activated(CodeActivated and PointerActivated)`
`Deactivated`	`OnPause`	`OnResignActivation`	`Activated (Deactivated)`
`Stopped`	`OnStop`	`DidEnterBackground`	`VisibilityChanged`
`Resumed`	`OnRestart`	`WillEnterForeground`	`Resumed`
`Destroying`	`OnDestroy`	`WillTerminate`	`Closed`

This list may not provide too much meaning right now, and I wouldn't worry yourself with needing to know this. The aim here is to provide you with a quick look-up to be able to then research if any lifecycle events are going wrong or possibly not the right fit for your solution. I can safely say that a large number of the issues I have helped clients with in the past are around how the lifecycle of an application differs on each platform supported by .NET MAUI.

Platform-Specific Lifecycle Events

There are actually many platform-specific lifecycle events that .NET MAUI does not map to. What .NET MAUI does provide is a set of lifecycle events that map consistently across all platforms. The rest in this section are really specific to each individual platform. I won't be covering all of the details of each individual event; however, I will cover how to make use of one so that you will know how to make use of an event that better suits your use case.

When searching for information around a platform-specific event, don't feel constrained to searching for .NET MAUI-specific documentation. You have the power to leverage the platform APIs. You should be able to search for information in the context of Android or iOS, and the code should be relatively easy to translate into C#.

In order to register for a platform-specific event, you need to make use of the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` class. Let's look at a concrete example for each platform. The code in each of the following examples is largely the same, but the duplication has been kept to show the bigger picture. I have highlighted the differences in **bold** to show the key differences.

Android

To receive a notification for an Android lifecycle event, you call the ConfigureLifecycleEvents method on the MauiAppBuilder object. You can then make use of the AddAndroid method and specify the events you wish to handle and how you wish to handle them.

```
using Microsoft.Maui.LifecycleEvents;
namespace WidgetBoard;
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureLifecycleEvents(events =>
            {
#if ANDROID
                events.AddAndroid(lifecycle=>
                    lifecycle.OnStart((activity) =>
                    OnStart(activity)));
                static void OnStart(Activity activity)
                {
                    // Perform your OnStart logic
                }
#endif
            });
        return builder.Build();
    }
}
```

For more information on the available lifecycle events, I recommend checking out the following documentation pages:

Microsoft: https://learn.microsoft.com/dotnet/maui/fundamentals/app-lifecycle#android

Android: https://developer.android.com/guide/components/activities/activity-lifecycle

iOS and MacCatalyst

To receive a notification for an iOS lifecycle event, you call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object. You can then make use of the `AddiOS` method and specify the events you wish to handle and how you wish to handle them.

```
using Microsoft.Maui.LifecycleEvents;
namespace WidgetBoard;
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureLifecycleEvents(events =>
            {
#if IOS || MACCATALYST
                events.AddiOS(lifecycle =>
                    lifecycle.OnActivated((app) =>
                    OnActivated(app)));
                static void OnActivated(UIKit.UIApplication
                application)
```

```
                    {
                        // Perform your OnActivated logic
                    }
#endif
                });
            return builder.Build();
        }
}
```

For more information on the available lifecycle events, I recommend checking out the following documentation pages:

Microsoft: https://learn.microsoft.com/dotnet/maui/fundamentals/app-lifecycle#ios

iOS: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle?language=objc

Windows

To receive a notification for a Windows lifecycle event, you call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object. You can then make use of the `AddWindows` method and specify the events you wish to handle and how you wish to handle them.

```
using Microsoft.Maui.LifecycleEvents;
namespace WidgetBoard;
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureLifecycleEvents(events =>
```

```
            {
#if WINDOWS
                events.AddWindows(lifecycle =>
                    lifecycle.OnActivated((window, args) =>
                    OnActivated(window, args)));
                static void OnActivated(Microsoft.
                UI.Xaml.Window window, Microsoft.UI.Xaml.
                WindowActivatedEventArgs args)
                {
                    // Perform your OnActivated logic
                }
#endif
            });
        return builder.Build();
    }
}
```

For more information on the available lifecycle events, I recommend checking out the following documentation page:

Microsoft: https://learn.microsoft.com/dotnet/maui/fundamentals/app-lifecycle#windows

You may have noticed the usage of #if statements. Due to the nature of compiling for multiple platforms in a single project, you will need to write platform-specific code. If, like me, you do not like the #if statement or you would like to keep its usage to a minimum, then fear not; we will be taking a closer look at minimizing it in Chapter 13.

Summary

In this chapter, you have

- Walked through the main components of a .NET MAUI application
- Earned a tea break
- Learned about the startup process
- Learned about the life of a .NET MAUI application

In the next chapter, you will

- Learn about the different possibilities you have to architect your applications
- Decide on what architecture to use
- Walk through a concrete example by creating your `ClockWidget`
- Learn how to further optimize your implementation using NuGet packages

CHAPTER 4

An Architecture to Suit You

Abstract

In this chapter, you will look through some possible architectural patterns that can be used to build .NET MAUI applications. The objective is to provide you with enough detail to help you find the architecture that best fits you. I want to point out that there are no right answers concerning which architecture to choose. The best option is to go with one that you feel will benefit you and your team.

I aim to quash the following myths throughout the course of this chapter:

> "You are forced to use XAML."

> "You are forced to use MVVM."

There seems to be a common misconception that .NET MAUI (and previously Xamarin.Forms) is built largely around using only XAML and MVVM. While this is the most common approach taken by developers, it is not forced upon us.

In order to compare some of the available architecture options, we will need something to compare; this leads us onto our measuring stick.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_4

A Measuring Stick

You will build the same control with each of the options to provide a way to compare the differences. The control you will be building is a `ClockWidget`. The purpose of this control is to do the following:

- Display the current time in your app.
- Update the time every minute.

Figure 4-1 shows a very rough layout of the control with the current date and time. You will tidy this up later with the ability to format the date and time information in Chapter 5, but for now, let's just focus on a limited example to highlight the differences in options. Figure 4-1 shows how the `ClockWidget` will render in your application when you have finished with this chapter.

Figure 4-1. *Sketch of how the ClockWidget control will render*

Prerequisites

Before you get started with each of the architectures you will be reviewing in this chapter, you need to do a little bit of background setup to prepare.

You need to add a single new class. This implementation will allow your widgets to schedule an action of work to be performed after a specific period of time. In your scenario of the `ClockWidget`, you can schedule an update of the UI. Let's add this `Scheduler` class into your project.

- Right-click the *WidgetBoard* project.
- Select **Add ➤ Class**.
- Give it the name of *Scheduler*.
- Click **Add**.

You want to modify the contents of the file to look as follows:

```
namespace WidgetBoard;

public class Scheduler
{
    public void ScheduleAction(TimeSpan timeSpan,
    Action action)
    {
        Task.Run(async () =>
        {
            await Task.Delay(timeSpan);
            action.Invoke();
        });
    }
}
```

In the following sections, you will be looking at code examples rather than implementing them directly. This is aimed at providing some comparisons to allow you to find out what will be a good fit for you as you build your applications and grow as a cross-platform developer. At the end of the chapter, you will take your chosen approach and add it into your application so you can see the final result of your `ClockWidget`.

Model View ViewModel (MVVM)

Model View ViewModel is a software design pattern that focuses on separating the user interface (View) from the business logic (Model). It achieves this with the use of a layer in between (ViewModel). MVVM allows a clean separation of presentation and business logic. Figure 4-2 shows the clean separation between the components of the MVVM architecture.

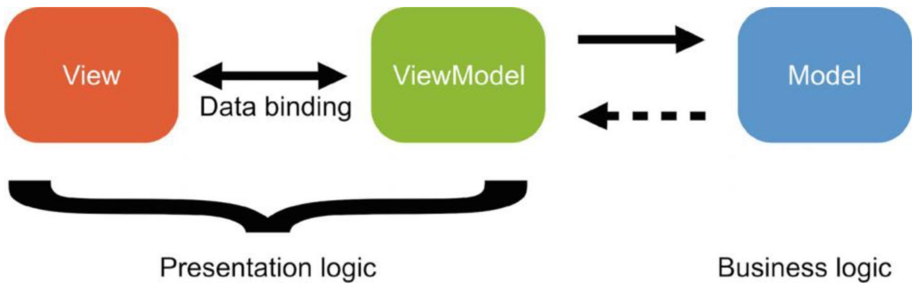

Figure 4-2. *An overview of the MVVM pattern*

The result of creating this separation between UI and business logic brings several benefits:

- Makes unit testing easier
- Allows for Views to be swapped out or even rewritten without impacting the other parts
- Encourages code reuse
- Provides the ability to separate UI development from the business logic development

A key part to any design pattern is knowing where to locate parts of your code to make it fit and abide by the rules. Let's take a deeper look at each of the three key parts of this pattern.

Model

The Model is where you keep your business logic. It is typically loaded from a database/web service among many other things.

For your business logic, you are going to rely on the `Scheduler` class that you created earlier in the "Prerequisites" section of this chapter.

View

The View defines the layout and appearance of the application. It is what the user will see and interact with. In .NET MAUI, a View is typically written in XAML where possible, but there will be occasions when logic in the code-behind will need to be written. You will learn this later in this chapter; you don't have to use XAML at all, so if you don't feel XAML is right for you, fear not.

A View in .NET MAUI is typically a `ContentPage` or an implementation that will inherit from `ContentPage` or `ContentView`. You use a `ContentPage` if you want to render a full page in your application (basically a view that will fill the application). You use a `ContentView` for something smaller (like a widget!). For your implementation, you will be inheriting from a `ContentView`.

I discussed in Chapter 2 that the majority of XAML files come with an associated C# file. A XAML-based view is no exception to this rule. With this in mind, let's take a look at the contents you need to place in each of the files.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentView
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.ClockWidget">
    <ContentView.BindingContext>
        <viewmodels:ClockWidgetViewModel />
    </ContentView.BindingContext>
    <Label Text="{Binding Time}"
```

```
            FontSize="80"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
</ContentView>
```

C# (Code-Behind)

The following code will have already been created for you by the .NET MAUI template. It is included for reference.

```
namespace WidgetBoard;
public partial class ClockWidget : ContentView
{
    public ClockWidget()
    {
        InitializeComponent();
    }
}
```

The `InitializeComponent` method call above is essential when building XAML-based views. It results in the XAML being loaded and parsed into an instance of the controls that have been defined in the XAML file.

ViewModel

The ViewModel acts as the bridge between the View and the Model. You expose properties and commands on the ViewModel that the View will bind to. To make a comparison to building applications with just code-behind, we could state that properties basically map to references of controls and commands are events. A binding provides a mechanism for both the View and ViewModel to send and receive updates.

For your ViewModel to notify the View that a property has changed and therefore the View will refresh the value displayed on screen, you need to make use of the INotifyPropertyChanged interface. This offers a single PropertyChanged event that you must implement and ultimately raise when your data-bound value has changed. This is all handled by the XAML binding engine, which you will look at in much more detail in the next chapter. Let's create your ViewModel class and then break down what is going on.

```
public class ClockWidgetViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private readonly Scheduler scheduler = new();
    private DateTime time;

    public DateTime Time
    {
        get
        {
            return time;
        }
        set
        {
            if (time != value)
            {
                time = value;
                PropertyChanged?.Invoke(this, new Property
                ChangedEventArgs(nameof(Time)));
            }
        }
    }
```

```
    public ClockWigetViewModel()
    {
        SetTime(DateTime.Now);
    }

    public void SetTime(DateTime dateTime)
    {
        Time = dateTime;
        scheduler.ScheduleAction(
            TimeSpan.FromSeconds(1),
            () => SetTime(DateTime.Now));
    }
}
```

You have

- Created a class called `ClockWidgetViewModel`
- Implemented the `INotifyPropertyChanged` interface
- Added a property that when `set` will check whether its value really has changed, and if it has, raise the `PropertyChanged` event with the name of the property that has changed
- Added a method to set the `Time` property and repeat every second so that the widget looks like a clock counting

Model View Update (MVU)

Model View Update is a software design pattern for building interactive applications. The concept originates from the Elm programming language. As the name suggests, there are three key parts to MVU:

- Model: This is the state of your application.
- View: This is a visual representation of your state.
- Update: This is a mechanism to update your state.

Figure 4-3 shows how each of these components relates and interacts with each other.

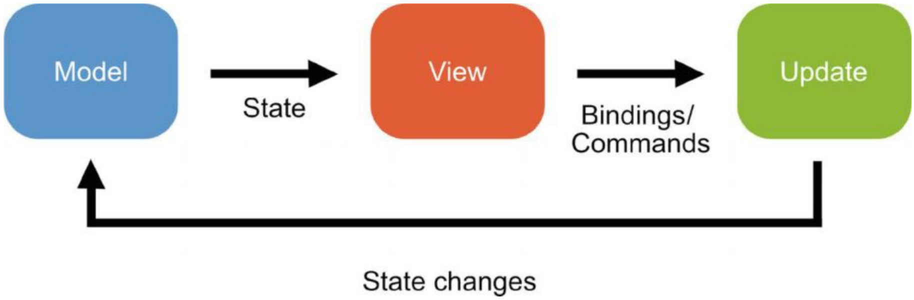

Figure 4-3. *An overview of the MVU pattern*

This pattern offers several benefits:

- Clearly defined rules around where state is allowed to be updated
- Ease of testing

A key part to any design pattern is knowing where to locate parts of your code to make it fit and abide by the rules. Let's take a deeper look at each of the three key parts of this pattern.

Getting Started with MauiReactor

It is worth noting that the MVU library that we will be using is not directly provided by Microsoft; instead, it is a community-based project called MauiReactor. The project can be found on GitHub at `https://github.com/adospace/reactorui-maui`.

First, you must install the MauiReactor project templates. To do this, open a terminal window and run the following command.

macOS

1. Open the **Terminal** application.
2. Enter the following command and then press return:

   ```
   dotnet new --install Reactor.Maui.TemplatePack
   ```

Windows

1. Open the **Command Prompt** application.
2. Enter the following command and then press return:

   ```
   dotnet new --install Reactor.Maui.TemplatePack
   ```

This will install the template so that you can create a new project. Sadly, this is different enough to the WidgetBoard project that you have been working with so far.

Next, you need to create the project. This is again done via the terminal for now:

```
dotnet new maui-reactor-startup --name WidgetBoard.Mvu
```

This will create a new project that you can start modifying.

Overview of the MVU Project Format

Let's have a quick overview of the project structure of the MVU-based project that was just created in order to become familiar. If you open the project you just created in Visual Studio or Rider, you will notice that the structure looks similar to a standard .NET MAUI project. One key difference is that there are very few (only two) XAML files; the bulk of the applications written in MVU will be through using C#.

The next key detail is how a View is represented. Views in MauiReactor are referred to as Components; the aim of building components is to create small reusable components that can make up the building blocks on an application or multiple applications. The Views are considered immutable in MVU, which means they will never update; instead, when the state (model) is updated, the view will be redrawn in order to visually represent the changes to the state. With this detail in mind, it is essential to build small components in order to limit the amount of the application that needs to be redrawn when some state changes.

Let's proceed to making some changes in order to see MVU in action in .NET MAUI. The template will have created a single *MainPage.cs* file under the */Pages* folder. This is the file that we are going to want to modify for the purpose of creating a ClockWidget.

Adding Your MVU Implementation

Go ahead and open the MainPage.cs file and make the following changes:

```
class MainPage : Component
{
    public override VisualNode Render()
        => ContentPage(
                new ClockWidget()
        );
}
```

The result of the above change will be to present a page with a single component inside.

Finally, go ahead and create your ClockWidget class:

```
public class ClockState
{
    public DateTime Time { get; set; }
}

public class ClockWidget : Component<ClockState>
{
    public override VisualNode Render()
    {
        return new VerticalStackLayout
        {
            Label(State.Time.ToString())
                .FontSize(80)
                .HCenter()
                .VCenter(),

            new Timer(interval: TimeSpan.FromSeconds(1), () =>
            SetState(s => s.Time = DateTime.Now))
                .IsEnabled(true)
        };
    }
}
```

Now that you have added a load of code, let's summarize what you have done.

- You have created your state (model) class ClockState.
- You have created a new component named ClockWidget.
- You have defined your state type as ClockState.

- You have initialized (known as `init` in the MVU pattern) your **model** field `clock`.
- You have defined the visuals of your component with the `Render()` function.
- You have added a Timer component that will update your state every second with the current date/time.

Note that there are two common scenarios when an update is called: when there is user interaction (e.g., a click/tap of a button) and around asynchronous background work. Your example here applies to the second scenario.

As I mentioned earlier, we can see how our clock widget can be achieved using MVU; this is a relatively simple example so I would strongly recommend checking out the more in-depth examples provided by the MauiReactor team at https://github.com/adospace/reactorui-maui.

XAML vs. C# Markup

XAML has proven to be a big part of building application UIs in .NET MAUI, but I want to make it clear that you do not have to use it. So if like some friends and colleagues, the verbosity of XAML makes you feel queasy, there is a solution!

Anything that you can create in XAML can ultimately be created in C#. Furthermore, there are ways to improve on the readability of the C# required to build UIs.

Some benefits of building user interfaces solely with C# are

- A single file for a view. No pairing of `.xaml.cs` and `.xaml` files.
- Better refactoring options so renaming properties or commands in XAML won't update the C#.

Let's work through how you can build your `ClockWidget` in C# in all its verbosity, and then I will show how you can simplify it using C# Markup. (I must add this is an open source package that you need to bring in.) Also, these examples are still built using MVVM.

Plain C#

As mentioned, anything you can build in XAML can also be built in C#. The following code shows how the exact same XAML definition of your `ClockWidget` can be built using just C#:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public class ClockWidget : ContentView
{
    public ClockWidget()
    {
        BindingContext = new ClockWidgetViewModel();
        var label = new Label
        {
            FontSize = 80,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
        label.SetBinding(
            Label.TextProperty,
            nameof(ClockWidgetViewModel.Time));
        Content = label;
    }
}
```

The code above does the following things:

- Creates a single file representing your `ClockWidget`
- Points your widget's `BindingContext` to the `ClockWidgetViewModel`
- Creates a label and sets its `Text` property to be bound to the view model's `Time` property
- Assigns the label to the content of the view

C# Markup

I have recently come to appreciate the value of being able to fluently build UIs. I don't tend to do it often because I personally feel comfortable building with XAML or perhaps it is Stockholm syndrome kicking in ☺ (I've been working with XAML for well over ten years now). When I do, it needs to be as easy to read and build as possible given it is not something I do often.

As a maintainer on the .NET MAUI Community Toolkit, one of the packages we provide is CommunityToolkit.Maui.Markup. It provides a set of extension methods and helpers to build UIs fluently.

```
using CommunityToolkit.Maui.Markup;
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public class ClockWidget : ContentView
{
    public ClockWidget()
    {
        BindingContext = new ClockWidgetViewModel();
        Content = new Label()
```

```
            .Font(size: 80)
            .CenterHorizontal()
            .CenterVertical()
            .Bind(Label.TextProperty, getter: static
            (ClockWidgetViewModel viewModel) =>
            viewModel.Time);
    }
}
```

This code performs the same steps as the plain C# example; however, the code is much easier to read. I am sure you can imagine that when the complexity of the UI increases, this fluent approach can really start to benefit you.

Chosen Architecture for This Book

Throughout this book, we will be using the MVVM-based architecture while building the UI through XAML.

My reasons for choosing MVVM are as follows:

- I have spent the last 10+ years using this architecture so it certainly feels natural to me.
- It has been a very common way of building applications over the past decade so there is an abundance of resources online to assist in overcoming issues around it.
- It is a common pattern in all Microsoft products and has a proven track record.

Now that I have covered the various architecture options and decided on using MVVM, let's proceed to adding in the specific Views and ViewModels so that they can be used inside the application. Then I will show how to start simplifying the implementation so that the code really only needs to include the core logic by avoiding having to add a lot of the boilerplate code.

Adding the ViewModels

First, add a new folder to your project.

- Right-click the *WidgetBoard* project.
- Select **Add ➤ New Folder**.
- Enter the name *ViewModels.*
- Click **Add**.

This folder will house your application's view models. Let's proceed to adding the first one.

Adding IWidgetViewModel

The first item you need to add is an interface. It will represent all widget view models that you create in your application.

- Right-click the `ViewModels` folder.
- Select **Add ➤ New Item**.
- Select the **Interface** type.
- Enter the name *IWidgetViewModel.*
- Click **Add**.

Modify this file to the following:

```
namespace WidgetBoard.ViewModels;
public interface IWidgetViewModel
{
    int Position { get; set; }
    string Type { get; }
}
```

Adding BaseViewModel

This will serve as the base class for all of your view models so that you only have to write some boilerplate code once. Don't worry; you will see how to optimize this even further!

- Right-click the `ViewModels` folder.
- Select **Add ➤ Class**.
- Enter the name *BaseViewModel.*
- Click **Add**.

You can replace the contents of the class file with the following code:

```
namespace WidgetBoard.ViewModels;

public abstract class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string
    propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEvent
        Args(propertyName));
    }
```

```
    protected bool SetProperty<TValue>(ref TValue backingField,
    TValue value, [CallerMemberName] string propertyName = "")
    {
        if (Comparer<TValue>.Default.Compare(backingField,
        value) == 0)
        {
            return false;
        }
        backingField = value;
        OnPropertyChanged(propertyName);
        return true;
    }
}
```

You should be familiar with the first line inside the class:

```
public event PropertyChangedEventHandler PropertyChanged;
```

This is the event definition that you must add as part of implementing the `INotifyPropertyChanged` interface, and it serves as the mechanism for your view model to update the view.

The next method provides a mechanism to easily raise the `PropertyChanged` event:

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

The `OnPropertyChanged` method can be called with or without passing in a value for `propertyName`. By passing a value in, you are indicating which property name on your view model has changed.

If you do not, then the [CallerMemberName] attribute indicates that the name of the caller will be used. Don't worry if this is a little unclear right now; it will become much clearer when you add your property into your ClockWidgetViewModel so just bear with me.

The final method adds a lot of value:

```
protected bool SetProperty<TValue>(
    ref TValue backingField,
    TValue value,
    [CallerMemberName] string propertyName = "")
{
    if (Comparer<TValue>.Default.Compare(backingField,
    value) == 0)
    {
        return false;
    }
    backingField = value;
    OnPropertyChanged(propertyName);
    return true;
}
```

The SetProperty method does the following:

- Allows you to call it from a property setter, passing in the field and value being set.
- Checks whether the value is different from the backing field, basically determining whether the property has really changed.
- If it has changed, it fires the PropertyChanged event using your new OnPropertyChanged method.
- Returns a Boolean indicating whether the value did really change. This can be really useful when needing to update other properties or commands!

This concludes the base view model implementation. Let's proceed to using it as the base for the ClockWidgetViewModel to really appreciate the value it is providing.

Adding ClockWidgetViewModel

Let's add a new class file into your ViewModels folder as you did for the BaseViewModel.cs file. Call this file ClockWidgetViewModel and modify the contents to the following:

```
using System;
using System.ComponentModel;

namespace WidgetBoard.ViewModels;

public class ClockWidgetViewModel : BaseViewModel,
IWidgetViewModel
{
    private readonly Scheduler scheduler = new();
    private DateTime time;

    public DateTime Time
    {
        get => time;
        set => SetProperty(ref time, value);
    }

    public int Position { get; set; }
    public string Type => "Clock";

    public ClockWidgetViewModel()
    {
        SetTime(DateTime.Now);
    }
```

```
    private void SetTime(DateTime dateTime)
    {
        Time = dateTime;
        scheduler.ScheduleAction(
            TimeSpan.FromSeconds(1),
            () => SetTime(DateTime.Now));
    }
}
```

The above code should be familiar. You saw it when reviewing MVVM. The optimization made here is to reduce the size of the `Time` property down to just 5 lines where the original example was 16 lines of code.

Adding Views

First, add a new folder to your project.

- Right-click the *WidgetBoard* project.
- Select **Add ➤ New Folder**.
- Enter the name *Views.*
- Click **Add**.

This folder will house your application's views. Let's proceed to adding your first one.

Adding IWidgetView

The first item you need to add is an interface to represent all widget view models that you create in your application.

- Right-click the `Views` folder.
- Select **Add ➤ New Item**.

- Select the **Interface** type.
- Enter the name *IWidgetView.*
- Click **Add**.

Modify the contents of this file to the following:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public interface IWidgetView
{
    int Position
    {
        get => WidgetViewModel.Position;
        set => WidgetViewModel.Position = value;
    }
    IWidgetViewModel WidgetViewModel { get; set; }
}
```

Adding ClockWidgetView

The next item you need to add is a `ContentView`. This is the first time you are doing this, so use the following steps:

- Right-click the `Views` folder.
- Select **Add ➤ New Item**.
- Select the **.NET MAUI** tab.
- Select the **.NET MAUI ContentView (XAML)** option.
- Enter the name *ClockWidgetView.*
- Click **Add**.

Observe that two new files have been added to your project: ClockWidgetView.xaml and ClockWidgetView.xaml.cs. You may notice that the ClockWidgetView.xaml.cs file is hidden in the Solution Explorer panel and that you need to expand the arrow to the left of the ClockWidgetView.xaml file.

Let's update both files to match what was in the original examples.

Open the ClockWidgetView.xaml file and modify the contents to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<Label
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Views.ClockWidgetView"
    FontSize="80"
    VerticalOptions="Center"
    HorizontalOptions="Center"
    x:DataType="viewModels:ClockWidgetViewModel"
    Text="{Binding Time}">
</Label>
```

Open the ClockWidgetView.xaml.cs file and modify the contents to the following:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public partial class ClockWidgetView : Label, IWidgetView
{
    public ClockWidgetView()
    {
        InitializeComponent();
```

```
        WidgetViewModel = new ClockWidgetViewModel();
        BindingContext = WidgetViewModel;
    }
    public IWidgetViewModel WidgetViewModel { get; set; }
}
```

This completes the work to add the `ClockWidget` into your code base. Now you need to modify your application so that you can see this widget in action!

Viewing Your Widget

In order to view your widget in your application, you need to make some changes to the `MainPage.xaml` and `MainPage.xaml.cs` files that were generated when you first created your project.

Modifying MainPage.xaml

Simply replace the contents of the file with the following:

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

The original file had a basic example that ships with the .NET MAUI template, but it wasn't of much use in this application.

Modifying MainPage.xaml.cs

You need to modify the contents of this file because you deleted some controls from the `MainPage.xaml` file. If you don't update this file, Visual Studio will report compilation errors. You can replace the entire contents of the `MainPage.xaml.cs` file with the following to remove references to the controls you deleted from the XAML file:

```
namespace WidgetBoard;

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

This concludes the changes that you need to make in your application. Let's see what your application looks like now!

Taking the Application for a Spin

If you build and run your application just like you learned to in Chapter 2, you can see that it renders the `ClockWidget` just as I originally designed. Figure 4-4 shows the clock widget rendered in the application running on macOS.

Figure 4-4. *The clock widget rendered in the application running on macOS*

You have looked at ways to optimize your code base when using MVVM, but I would like to provide some further details on how you can leverage the power of the community in order to further improve your experience.

MVVM Enhancements

There are two key parts I will cover regarding how you can utilize existing packages to reduce the amount of code you are required to write.

MVVM Frameworks

There are several MVVM frameworks that can expand on this by providing a base class implementation for you with varying levels of other extra features. To list a few:

- CommunityToolkit.Mvvm
- FreshMVVM
- Prism
- ReactiveUI

These packages will ultimately provide you with a base class very similar to the `BaseViewModel` class that you created earlier. For example, the Prism library provides the `BindableBase` class that you could use. It offers yet another optimization in terms of less code that you need to write and ultimately maintain.

You can go a step further, but you need to believe.

Magic

Yes, that's right: magic is real! These approaches involve auto-generating the required boilerplate code so that we as developers do not have to do it. There are two main packages that offer this functionality. They provide it through different mechanisms, but they work equally well.

- Fody: IL generation, `https://github.com/Fody/Home`
- CommunityToolkit.Mvvm: Source generators (yes, this gets a second mention), `https://learn.microsoft.com/dotnet/communitytoolkit/mvvm/`

In the past, I was skeptical of using such packages. I felt like I was losing control of parts that I needed to hold on to. Now I can appreciate that I was naïve, and this is impressive.

Let's look at how these packages can help to further reduce the code. This example uses CommunityToolkit.Mvvm, which provides the `ObservableObject` base class and a wonderful way of adding attributes (`[ObservableProperty]`) to the fields you wish to trigger `PropertyChanged` events when their value changes. This will then generate a property with the same name as the field but with a capitalized first character, so `time` becomes `Time`.

```
public partial class ClockWidgetViewModel : ObservableObject
{
    [ObservableProperty]
    private DateTime time;
```

```
    public ClockWigetViewModel()
    {
        SetTime(DateTime.Now);
    }

    public void SetTime(DateTime dateTime)
    {
        Time = dateTime;
        scheduler.ScheduleAction(
            TimeSpan.FromSeconds(1),
            () => SetTime(DateTime.Now));
    }
}
```

That's 17 lines down to 2 from the original example! The part that I really like is that it reduces all the noise of the boilerplate code so there is a bigger emphasis on the code that we need to write as developers.

You may have noticed that you are still referring to the `Time` property in the code but you haven't supplied the definition for this property. This is where the magic comes in! If you right-click the `Time` property and select *Go to Definition…*, it will open the following source code so you can view what the toolkit has created for you:

```
// <auto-generated/>
#pragma warning disable
#nullable enable
namespace WidgetBoard.ViewModels
{
    partial class ClockWidgetViewModel
    {
        /// <inheritdoc cref="time"/>
```

```
[global::System.CodeDom.Compiler.GeneratedCode
("CommunityToolkit.Mvvm.SourceGenerators.
ObservablePropertyGenerator", "8.0.0.0")]
[global::System.Diagnostics.CodeAnalysis.
ExcludeFromCodeCoverage]
public global::System.DateTime Time
{
    get => time;
    set
    {
        if (!global::System.Collections.Generic.
        EqualityComparer<global::System.DateTime>.
        Default.Equals(time, value))
        {
            OnTimeChanging(value);
            OnPropertyChanging(global::Community
            Toolkit.Mvvm.ComponentModel.__Internals.__
            KnownINotifyPropertyChangingArgs.Time);
            time = value;
            OnTimeChanged(value);
            OnPropertyChanged(global::CommunityTool
            kit.Mvvm.ComponentModel.__Internals.__
            KnownINotifyPropertyChangedArgs.Time);
        }
    }
}
/// <summary>Executes the logic for when <see
cref="Time"/> is changing.</summary>
[global::System.CodeDom.Compiler.GeneratedCode
("CommunityToolkit.Mvvm.SourceGenerators.
ObservablePropertyGenerator", "8.0.0.0")]
```

```
        partial void OnTimeChanging(global::System.
        DateTime value);
        /// <summary>Executes the logic for when
        <see cref="Time"/> just changed.</summary>
        [global::System.CodeDom.Compiler.GeneratedCode
        ("CommunityToolkit.Mvvm.SourceGenerators.
        ObservablePropertyGenerator", "8.0.0.0")]
        partial void OnTimeChanged(global::System.
        DateTime value);
    }
}
```

You can see that the generated source code looks a little noisy, but it does in fact generate the property you need. View the section highlighted in **bold** above.

I have only really scratched the surface regarding the functionality that the CommunityToolkit.Mvvm offers. I strongly urge you to refer to the documentation at https://learn.microsoft.com/dotnet/communitytoolkit/mvvm/ to learn how it can further aid your application development because this will not be looked into any deeper in this book so we can focus on the fundamentals.

Summary

I hope I have made it clear that there is no single right way to do things or build applications. You should pick and choose what approaches will best suit your environment. With this point in mind, the goal of this chapter was to give you a good overview of several different approaches to architecting your application. There are always a lot of opinions floating around to indicate which architectures people prefer, but I strongly urge you to evaluate which will help you to achieve your goals best.

In this chapter, you have

- Learned about the different possibilities you have to architect your applications
- Decided on what architecture to use
- Walked through a concrete example by creating the `ClockWidget`
- Learned how to further optimize your implementation using NuGet packages

In the next chapter, you will

- Create and apply an icon in your application
- Add some placeholder pages and view models
- Fill your first page with some UI and set up bindings to the view model
- Explore data binding and its many uses
- Gain an understanding of XAML
- Learn about the possible layouts you can use to group other controls
- Gain an understanding of Shell and apply this to building your application's structure
- Apply the Shell navigation to allow you to navigate
- Build your flyout menu

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch04`.

CHAPTER 5

User Interface Essentials

Abstract

In this chapter, you are going to investigate the fundamental parts of building a .NET MAUI application. You are going to apply an icon and splash screen, add in some pages and their associated view models, and configure some bindings between your page and the view model. You will also gain an understanding of what XAML is and what it has to offer as you build the pages of your application.

Prerequisites

You need to do some setup before you can jump into using Shell. If Shell is still feeling like an unknown concept, fear not. I will be covering it a little bit later in this chapter under the "Shell" section.

Let's go ahead and add the following folders to your project.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_5

Models

This will house all of your Model classes. If you recall from Chapter 4, these are where some of your business logic is located. In your `Models` folder, you need to create one class.

- Right-click the *Models* folder.
- Select **Add ➤ New Class**.
- Click **Add**.

Board.cs

This will serve as a base class for the layout options you provide. In this book, you will only be building fixed layout boards, but I wanted to lay some groundwork so if you are feeling adventurous, you can go off and build alternative layout options without having to restructure the application. In fact, I would love to hear where you take it!

Your fixed layout will offer the user of the app the ability to choose a number of rows and columns and then position their widgets in them.

```
namespace WidgetBoard.Models;

public class Board
{
    public string Name { get; init; } = string.Empty;
    public int NumberOfColumns { get; init; }
    public int NumberOfRows { get; init; }
}
```

This is the first time that we have used the `init` keyword in this book. I wanted to explain its use in case you are not familiar with it; the `init` keyword allows us to define a property that can be set only when a new instance is initialized. This means that the following is allowed:

```
var board = new Board
{
   Name = "Fixed Board";
   NumberOfColumns = 3;
   NumberOfRows = 3;
};
```

To highlight the value of the `init` keyword, the following code will generate three compiler errors, one compiler per property that hasn't been assigned a value.

```
var board = new Board();
```

Pages

This will house the pages in your application. I am distinguishing between a page and a view because they do behave differently in .NET MAUI. You can think of a page as a screen that you are seeing whereas a view is a smaller component. A page can contain multiple views.

Let's go ahead and create the following files under the `Pages` folder. The following steps show how to add the new pages.

- Right-click the *Pages* folder.
- Select **Add ➤ New Item**.
- Select the **.NET MAUI** tab.
- Select **.NET MAUI ContentPage (XAML)**.
- Click **Add**.

BoardDetailsPage

This is the page that lets you both create and edit your boards. For now, you will not touch the contents of this file. Note that you should see `BoardDetailsPage.xaml` and `BoardDetailsPage.xaml.cs` files created.

You also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method just before the `return builder.Build();` line.

```
builder.Services.AddTransient<BoardDetailsPage>();
```

ViewModels

This houses your ViewModels that are the backing for both your Pages and Views. You created this folder in the previous chapter, but you need to add a number of classes. The following steps show how to add the new pages:

- Right-click the *ViewModels* folder.
- Select **Add ➤ New Class**.
- Click **Add**.

BoardDetailsPageViewModel

This serves as the view model for the `BoardDetailsPage` file you created.

```
namespace WidgetBoard.ViewModels;

public class BoardDetailsPageViewModel : BaseViewModel
{
}
```

You also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method as you did above.

```
builder.Services.AddTransient<BoardDetailsPageViewModel>();
```

You should start to notice a common pattern with the creation of these files and the need to add them to the `MauiProgram.cs` file. This is to allow you to fully utilize the dependency injection provided by the framework, which you learned about in Chapter 3.

This concludes the prerequisite work required for this chapter, so let's proceed to covering the user interface essentials.

App Icons

Every application needs an icon, and for many people, this will be how they obtain their first impression. Thankfully these days device screens allow for bigger icon sizes and therefore more detail to be included in them.

As with general image resources, each platform requires different sizes and many more combinations to be provided. For example, iOS expects the following:

- Five different sizes of the app icon
- Three different sizes for the Spotlight feature
- Three different sizes for Notifications
- Three different sizes for Settings

That's up to 14 different image sizes required just for your application icon on iOS alone. See `https://developer.apple.com/design/human-interface-guidelines/ios/icons-and-images/app-icon/`.

.NET MAUI manages the process of generating all the required images for you. All you need to do is provide an SVG image file. Since SVGs are vector based, they can scale to each required size.

Adding Your Own Icon

Figure 5-1 shows the icon that you will be using for your application. You can grab a copy of the files that you will be using from https://github.com/bijington/introducing-dotnet-maui/tree/main/chapter05 and place them in the Resources/AppIcon folder. You should notice that they replace two existing files.

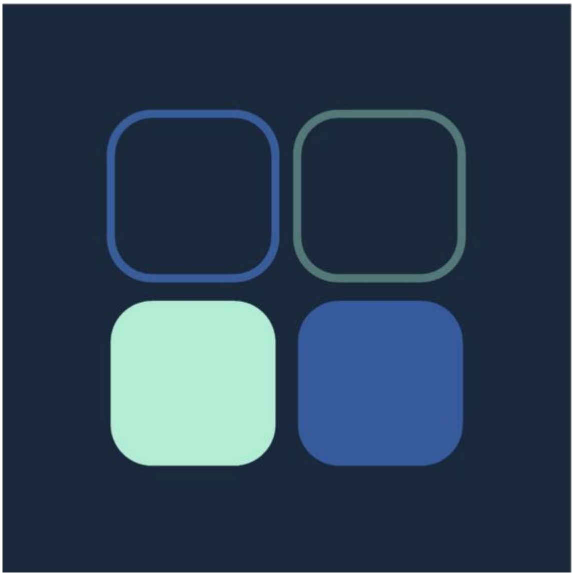

Figure 5-1. *Your application icon*

If you look in the contents of your project file, you will see the following entry:

```
<MauiIcon Include="Resources\AppIcon\appicon.svg" />
```

This tells the tooling to use the file appicon.svg and convert it into all the required sizes for each platform when building. Note you only want one MauiIcon in your project file. If you have multiple, the first one will be used.

You do not need to replace the above entry as the file you should have downloaded should have the name `appicon.svg`. If the file name is different, either rename it or update the name in the project file.

Platform Differences

It is worth noting that some platforms apply different rules to app icons and also can provide rather different outputs.

Android

App icons on Android can take many different shapes due to the different device manufacturers and their own flavor of the Android operating system. To cater for this, Google introduced the *adaptive icon*. This allows a developer to define two layers in their icon:

- The background: This is typically a single color or consistent pattern. It is the *appicon.svg* file that you downloaded.
- The foreground: This includes the main detail. It is the *appiconfig.svg* file that you downloaded.

.NET MAUI allows you to support the adaptive icon using the `IncludeFile` and the `ForegroundFile` properties on the `MauiIcon` element. You can see the `IncludeFile` is already defined in your project. This represents the background. You can split your application icon into two parts and then provide the detail to the `ForegroundFile`. Note that this can be applied to all platforms and is my recommended way to ship an application icon.

iOS and macOS

Apple does not allow for any transparency in an app icon. You can either make sure that you supply an image with no transparent pixels or you can use the `Color` property on the `MauiIcon` element, which will fill in any transparent pixels with that defined color.

Splash Screen

A splash screen is the first thing a user sees when they start your application. It gives you as a developer a way of showing the user something while the application is launching. Once everything has finished loading, the splash screen will be hidden and your main page will be shown.

In a similar manner to how the app icon is managed, the splash screen also has an entry in the project file and can generate a screen based on an SVG file. In fact, you will be using the same image to save effort.

```
<MauiSplashScreen Include="Resources\Splash\splash.svg"
Color="#512BD4" BaseSize="128,128" />
```

Note that splash screens built in this manner must be static. You can't have any animations running to show progress.

The `Color` property enables you to define a background color for the splash screen.

I have designed a splash screen image that you are free to use in your application, you can find a copy at https://github.com/bijington/introducing-dotnet-maui/tree/main/chapter05/splash and place them in the `Resources/Splash` folder.

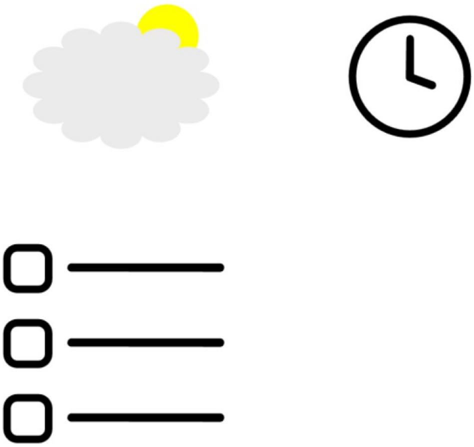

Figure 5-2. *Your application splash screen*

XAML

As a .NET MAUI developer, you will hear XAML mentioned many times; XAML stands for *eXtensible Application Markup Language*. It is an XML-based language used for defining user interfaces. It originates from WPF and Silverlight, but the .NET MAUI version has its differences.

There are two different types of XAML files that you will encounter when building your application:

- A `ResourceDictionary`: This is a single file that contains resources that can easily be used throughout your application. `Resources/Styles/Styles.xaml` is a perfect example of this. The `Styles.xaml` file is a default set of styles that is provided when you create a new .NET MAUI application. If you wish to modify some built-in styling, this is a very good place to do so.
- A View-based file: This contains both a `.xaml` and `.xaml.cs` file. They are paired together using the `partial` class keyword.

When dealing with this second item, you have to make sure that the `InitializeComponent` line is called inside the constructor; otherwise, the XAML will not be interpreted correctly, and you will see an exception thrown.

It is worth noting that XAML does not provide a rich set of features like C# does, and for this reason, there is almost always a `xaml.cs` file that goes alongside the XAML file. This C# file provides the ability to use the rich feature set of the C# language when XAML does not. For example, handling a button interaction event would have to be done within the C# code file.

Dissecting a XAML File

In the “Prerequisites” section of this chapter, you created the `BoardDetailsPage.xaml` file. Now you are going to modify it and add some meaningful content so you can start to see your application take shape. The code you should see in this file is shown below:

```
<?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="WidgetBoard.Pages.BoardDetailsPage"
    Title="BoardDetailsPage">
    <VerticalStackLayout>
        <Label
            Text="Welcome to .NET MAUI!"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </VerticalStackLayout>
</ContentPage>
```

If you break this down into small chunks, you can start to understand not only what makes up the UI of your application but also some of the fundamentals of how XAML represents it.

The root element is a `ContentPage`. As mentioned, a typical view in .NET MAUI is either a `ContentPage` or `ContentView`. As the name implies, it is a page that presents its content, and this will be a single view as its content.

As mentioned, XAML is an XML-based language, and there are the following key parts to understanding XAML:

1. Properties are set by attributes on your element, so

   ```
   <Label Text="Welcome to .NET MAUI!" />
   ```

   is effectively the same as writing

   ```
   new Label
   {
       Text = "Welcome to .NET MAUI!"
   };
   ```

2. XAML represents the visual hierarchy in the file structure. You can work out that `ContentPage` has a child of `VerticalStackLayout` and it has a child of `Label`. This can be especially helpful. A complex XAML file will result in a complex visual tree, and you want to try your best to avoid this because the greater the complexity results in poorer performance because the device will ultimately have to render more things on screen.

3. The `xmlns` tag works like a `using` statement in C#. This allows you to refer to other functionality that might not be available out of the box. For example, you can add the line `xmlns:views="clr-namespace:WidgetBoard.Views"` and it is the equivalent of adding `using WidgetBoard.Views;` in a C# file. This allows you to refer to the views in your code base.

The content of your `ContentPage` in your XAML is a `VerticalStackLayout`. I will cover layouts a little bit later in this chapter, but as a very brief overview, they allow you to have multiple child views as content and therefore open up the possibilities of creating your UIs. It is worth noting that a `ContentPage` can only have a single child, which makes layouts really important controls for use when building user interfaces.

Now that you have covered some of the key concepts around XAML, let's go ahead and start building your application's first page.

Building Your First XAML Page

I always like to work with a clear definition of what needs to be achieved so let's define what your page needs to do. It needs to do the following:

- Allow the user to create a new board.
- Fit on a variety of screen sizes.
- Allow the user to provide a name for the board.
- Allow the user to choose the layout type.
- Apply any valid properties for the specific layout type chosen.

Now that you know what needs to be achieved, let's go ahead and do it. You need to delete the existing contents of the page and replace them with a `Border`. A `Border` is similar to a `ContentView` in that it can only have a single child, but it offers you some extra properties that allow you to provide a nice looking UI. In particular, you care about the `StrokeShape` and `Stroke` properties. You may notice that you are not actually setting these properties in the XAML and you would be correct! There are two main reasons for this:

- You have suitable defaults defined in the `Resources/Styles/Styles.xaml` file that was created for you. Note that if you want to override these, it's perfectly fine. I will be covering this a little bit later in this chapter in the "Styling" section.
- It is considered good practice to only define the properties that you need to supply, which is basically anything that changes from the defaults. While the XAML compiler does a decent job of generating a UI that is defined at compile time, some bits are still potentially interpreted at runtime and this has a performance impact.

```
<?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="WidgetBoard.Pages.BoardDetailsPage">
    <Border
        MinimumWidthRequest="300"
```

```
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Padding="0">
            </Border>
        </ContentPage>
```

The most important parts of the properties that you are setting are the `HorizontalOptions` and `VerticalOptions`. They allow you to define where in the parent this view will be displayed. By default, a view will `fill` its parent's content, but you are going to make it float in the `center`. The main reason is so it will stay there regardless of the screen size it is running on. Of course, there are more in-depth ways of handling different screen sizes and you will explore them in the coming chapters.

While you have much more content to add to this XAML file, you are going to do so in the context of the following topics. Your next step is to add multiple child views. For this, you are going to need to choose a suitable Layout.

Layouts

.NET MAUI provides you with a set of prebuilt layout classes that allow you to group and arrange views in your application. The aim of this section is to explore each layout control and how it might be used for your application. I strongly recommend playing around with each of the layouts to see what will fit best for each individual use case and always remember to keep the visual tree as simple as possible.

AbsoluteLayout

As the name suggests, the `AbsoluteLayout` allows the positioning of its children with absolute values. The x, y, width, and height of a child are controlled through the `LayoutBounds` attached property. This means you use as follows

```
<AbsoluteLayout>
    <Label
        AbsoluteLayout.LayoutBounds="0,0,600,200"/>
</AbsoluteLayout>
```

Figure 5-3 shows how a control is positioned inside an `AbsoluteLayout`.

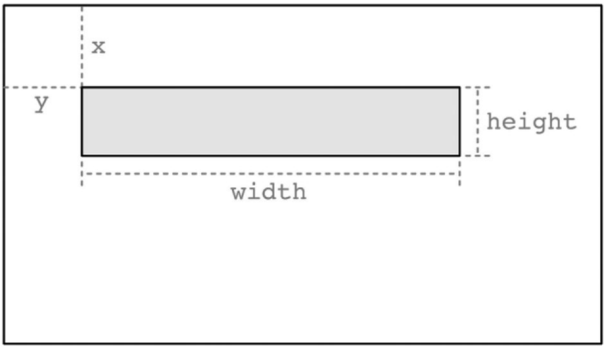

Figure 5-3. *AbsoluteLayout overview*

There is also the option to define layout bounds that are proportional to the `AbsoluteLayout` itself. You can control this with the `AbsoluteLayout.LayoutFlags` attached property.

```
<AbsoluteLayout>
    <Label
        AbsoluteLayout.LayoutBounds="0,0,0.5,0.2"
        AbsoluteLayout.LayoutFlags="All"/>
</AbsoluteLayout>
```

This will result in the `Label` being positioned at 0,0, but the width will be 50% of the `AbsoluteLayout` and the height will be 20%. This provides a lot of power when defining a user interface that can grow as the size of a device also increases.

The LayoutFlags option provides you with a lot of power. You can choose which part of the LayoutBounds is applied absolutely and which is applied proportionally. Here are the possible values for LayoutFlags and what they impact:

Value	Description
None	All values are absolute.
XProportional	The X property is proportional to the AbsoluteLayout dimensions.
YProportional	The Y property is proportional to the AbsoluteLayout dimensions.
WidthProportional	The Width property is proportional to the AbsoluteLayout dimensions.
HeightProportional	The Height property is proportional to the AbsoluteLayout dimensions.
PositionProportional	The X and Y properties are proportional to the AbsoluteLayout dimensions.
SizeProportional	The Width and Height properties are proportional to the AbsoluteLayout dimensions.
All	All properties are proportional to the AbsoluteLayout dimensions.

The AbsoluteLayout can be an incredibly powerful layout when used in the right scenario. For your scenario, it offers more complexities than I really think you need to handle.

FlexLayout

The FlexLayout comes with a large number of properties to configure how its children are positioned. If you want your controls to wrap, this is the control for you! A good example for using the FlexLayout is a media gallery.

Figure 5-4 shows how controls can be positioned inside a FlexLayout.

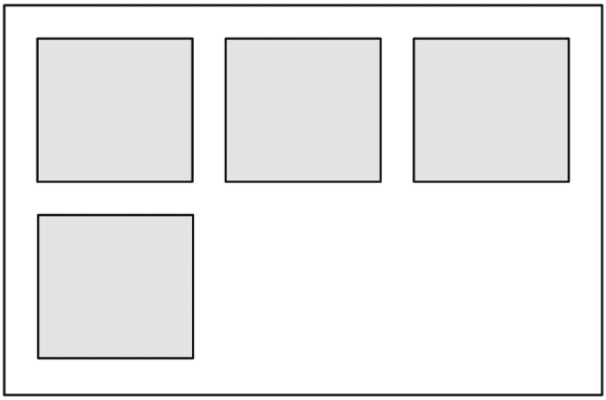

Figure 5-4. *FlexLayout overview*

The above layout can be achieved with the following code example:

```
<FlexLayout
    AlignItems="Start"
    Wrap="Wrap"
    Margin="30"
    JustifyContent="SpaceEvenly">
    <Border
        BackgroundColor="LightGray"
        WidthRequest="100"
        HeightRequest="100" />
    <Border
        BackgroundColor="LightGray"
```

```
        WidthRequest="100"
        HeightRequest="100" />
    <Border
        BackgroundColor="LightGray"
        WidthRequest="100"
        HeightRequest="100" />
    <Border
        BackgroundColor="LightGray"
        WidthRequest="100"
        HeightRequest="100" />
</FlexLayout>
```

Each of the properties you are using allows you to customize where each item is positioned during the rendering process and how it will move around in the application if it is resized. For further information on the possible ways of configuring the FlexLayout, read the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/user-interface/layouts/flexlayout.

Your BoardDetailsPage only needs controls positioned vertically so a FlexLayout feels like an overly complicated layout for this purpose.

Grid

I love Grids. They are usually my go-to layout option, mainly because I have become used to thinking about how they lay out controls and because they tend to allow you to keep your visual tree depth shallow. The layout essentially works by allowing you to define a set of rows and columns and then define which control should be displayed in which row/column combination.

Figure 5-5 shows how controls can be positioned inside a Grid.

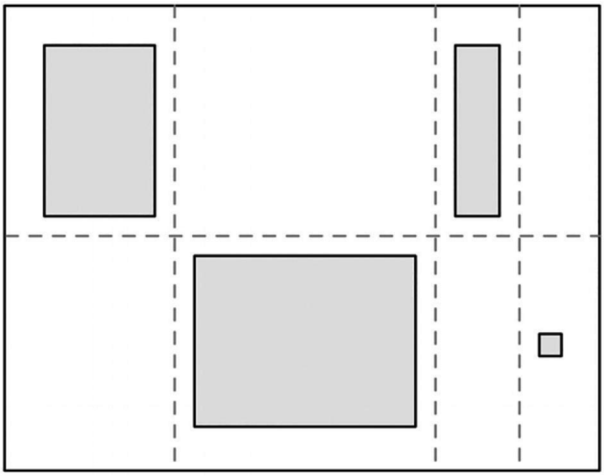

Figure 5-5. *Grid layout overview*

Controls inside a Grid are allowed to overlay each other, which can provide an extra tool in a developer's toolbelt when needing to show/hide controls. Controls in the Grid are arranged by first defining the ColumnDefinitions and RowDefinitions. Let's take a look at how to create the above layout with a Grid.

```
<Grid
    ColumnDefinitions ="*,2*,250,Auto"
    ColumnSpacing="20"
    Margin="30"
    RowDefinitions="*,*"
    RowSpacing="20">
    <Border
        BackgroundColor="LightGray"
        Grid.Column="0"
        Grid.Row="0" />
    <Border
        BackgroundColor="LightGray"
        Grid.Column="1"
```

```
        Grid.Row="1" />
    <Border
        BackgroundColor="LightGray"
        Grid.Column="2"
        Grid.Row="0" />
    <Border
        BackgroundColor="LightGray"
        Grid.Column="3"
        Grid.Row="1"
        WidthRequest="30"
        HeightRequest="30" />
</Grid>
```

You can see that you have created columns using a variety of different options:

- `250`: This is a fixed width of 250.
- `Auto:` This means that the column will grow in width based on its contents. It is recommended to use this option sparingly as it will result in the Grid control having to measure its children and force a rerender of itself and the other children.
- `*`: This is proportional and will result in the leftover space being allocated out. In this example, two columns use the * notation. This results in those two columns being allocated one-third and two-thirds of the remaining width, respectively. This is because * is actually considered 1*.

In your scenario, you are going to need multiple groups of controls. For this reason, I believe Grids will just make it slightly more complicated for you.

HorizontalStackLayout

The name really gives this away. It positions its children horizontally. The `HorizontalStackLayout` is not responsible for providing sizing information to its children, so the children are responsible for calculating their own size.

Figure 5-6 shows how controls can be positioned inside a `HorizontalStackLayout`.

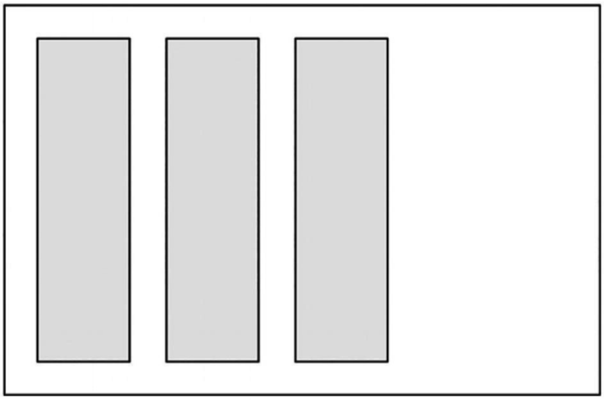

Figure 5-6. *HorizontalStackLayout overview*

The above layout can be achieved with the following code example:

```
<HorizontalStackLayout
    Spacing="20"
    Margin="30">
    <Border
        BackgroundColor="LightGray"
        WidthRequest="100" />
    <Border
        BackgroundColor="LightGray"
        WidthRequest="100" />
    <Border
```

```
        BackgroundColor="LightGray"
        WidthRequest="100" />
</HorizontalStackLayout>
```

You wish to layout your controls vertically so you can guess where this is going, although you will actually use one to group some of your inner controls.

VerticalStackLayout

The name really gives this away. It positions its children vertically. The `VerticalStackLayout` follows the same sizing rules as the `HorizontalStackLayout`, so the children are responsible for calculating their own size.

And there you have it: something that arranges its children vertically, which is exactly what you need!

Figure 5-7 shows how controls can be positioned inside a `VerticalStackLayout`.

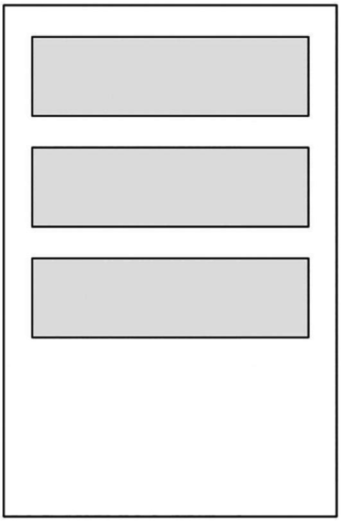

Figure 5-7. *VerticalStackLayout overview*

The above layout can be achieved with the following code example:

```
<VerticalStackLayout
    Spacing="20"
    Margin="30">
    <Border
        BackgroundColor="LightGray"
        HeightRequest="100" />
    <Border
        BackgroundColor="LightGray"
        HeightRequest="100" />
    <Border
        BackgroundColor="LightGray"
        HeightRequest="100" />
</VerticalStackLayout>
```

We mentioned that this is the layout that you will want to use in your page; let's go ahead and use it. Inside the `Border` you added earlier, add the following to your `BoardDetailsPage.xaml` file.

```
<VerticalStackLayout>
    <VerticalStackLayout
        Padding="20">
        <Label
            Text="Name"
            FontAttributes="Bold" />
        <Entry />
        <Label
            Text="Layout"
            FontAttributes="Bold" />
        <HorizontalStackLayout>
            <RadioButton
                x:Name="FixedRadioButton"
```

```
                Content="Fixed" />
        </HorizontalStackLayout>
        <VerticalStackLayout>
            <Label
                Text="Number of Columns"
                FontAttributes="Bold" />
            <Entry Keyboard="Numeric" />
            <Label
                Text="Number of Rows"
                FontAttributes="Bold" />
            <Entry Keyboard="Numeric" />
        </VerticalStackLayout>
    </VerticalStackLayout>
    <Button
        Text="Save"
        HorizontalOptions="End" />
</VerticalStackLayout>
```

Yes, I know! I spoke about keeping the visual tree simple and here you are nesting quite a few layouts. I find there is typically some level of pragmatism that needs to be applied. This page is still relatively simple in terms of what is being rendered on screen so I will argue that it is fine. If you were to repeat this layout multiple times, you would need to be a little more strict and find the best way to lay it all out. Quite often you will find that there can be a balancing act between defining something to give the best performance and making it easier to maintain as a developer.

So you have now built your UI, but you will notice that it doesn't do anything other than let the user type in the entry fields. You need to bind the view up to your view model.

This is not strictly part of layouts, but it is worth noting how you apply the `Keyboard` property to your `Entry` controls. This allows you to inform the operating system what soft keyboard to display and therefore limit the type of data the user can enter. Note that this only applies to mobile applications and it only really helps if a hardware keyboard is not used; if a user does connect a hardware keyboard, they will be able to enter invalid characters; therefore, it will still be up to us as developers to validate that the correct data has been entered. We will cover how to validate data in a reusable way in Chapter 9.

Data Binding

UI-based applications, as their name suggests, involve presenting an interface to the users. This UI is rarely ever just a static view and therefore needs to be updated, drive updates into the application, or both. This process is typically an event-driven one as either side of this synchronization needs to be notified when the other side changes. .NET MAUI wraps this process up for you through a concept called *data binding*. Data binding provides the ability to link the properties from two objects so that changes in one property are automatically updated in the second.

Binding

The most common type of bindings that you create is between a single value at the source and a single value at the target. The target is the owner of the bindable property. I use the terms *target* and *source* because you do not have to solely bind between a view and a view model. There are scenarios where you may wish to bind one control to another.

Before you jump into creating your first binding, you need to first create something to bind to. Open your `BoardDetailsPageViewModel` class, which is the view model for your view, and add the following:

```
private string boardName = string.Empty;
public string BoardName
{
    get => boardName;
    set => SetProperty(ref boardName, value);
}
```

It is worth noting that a `Binding` must be created against a property (e.g., the `BoardName` definition from the code above). Binding to a field (e.g., `boardName`) will not work.

BindingContext

And finally the crucial step is to set the `BindingContext` of your page to this view model. In Chapter 4, you did this by setting it in the XAML directly, but because you have registered your view model with the dependency injection layer, you can make the most of that and have it create the view model and whatever dependencies it has for you. Open your `BoardDetailsPage.xaml.cs` file and change the constructor to

```
public BoardDetailsPage(BoardDetailsPageViewModel
boardDetailsPageViewModel)
{
    InitializeComponent();
    BindingContext = boardDetailsPageViewModel;
}
```

The above code allows you to rely on the constructor injection functionality that .NET MAUI and Shell provide.

The act of setting the `BindingContext` property means that any bindings created in the page/view and any child views will be by default against this `BindingContext`.

Now if you jump into the `BoardDetailsPage.xaml` file, you can apply the binding to your new `BoardName` property in your view model. You want to modify the first `Entry` that you added to look like

```
<Entry Text="{Binding BoardName}" />
```

This is a relatively small change and will look like the bindings you created back in Chapter 4 when exploring the MVVM pattern. There isn't much detail to this, but there is a fair amount of implicit behavior that I feel I must highlight. Let's cover what it tells you first and then what it doesn't.

You are creating a binding between the `BoardName` property (which exists on your `BoardDetailsPageViewModel`) and the `Text` property on the `Entry` control.

Now on to what this code doesn't tell you.

Path

The binding could also be written as

```
Text="{Binding Path=BoardName}"
```

The `Path` element of the binding is implied if you do not explicitly provide it but only as the first part of the binding definition. Why am I telling you this? There are times when you will need to supply the `Path=` part.

Mode

I mentioned that bindings keep two properties in sync with each other. When you create a binding, you can define which direction the updates flow. In your example, you have not provided one, which then relies on the default `Mode` for the bindable property that you are binding to. In this case, it is the `Text` property of the `Entry`, which has a default binding mode of `TwoWay`. I strongly urge you to make sure you are aware of both these defaults and your expectation when creating a binding. Choosing the correct `Mode` can also boost performance. For example, the `OneTime` binding mode means that no updates need to be monitored for. In your scenario, you don't currently need to allow the view model to update the `Entry Text` property; however, as you progress, this page will also allow for the editing of a board so you will leave it alone. If you didn't need to edit, you could in theory modify your binding to be `Text="{Binding Path=BoardName, Mode=OneWay}"`.

There are several variations for binding modes:

- Default: As the name suggests, it uses the default, which is defined in the target property.
- TwoWay: It allows for updates to flow both ways between source and target. A typical example is binding to the `Text` property of an `Entry` where you want to both receive input from the user and update the UI, such as your scenario that you just added with the `Entry` and its `Text` property as `Text="{Binding Path=BoardName}"`.
- OneWay: It allows for updates to flow from the source to the target. An example of this is your ClockWidget where you only want updates to flow from your source to your target.
- OneWayToSource: It allows for updates to flow from the target to the source. An example of this is binding the `SelectedItem` property on the `ListView` to a value in your view model.

- OneTime: It only updates the target once when the binding context changes.

Source

As mentioned, a binding does not have to be created against something defined in your code (e.g., a property on a view model). It can, in fact, be created against another control. If you look back at the XAML you created for this page, you will notice that you gave the RadioButton the name of FixedRadioButton. This was actually setting you up for this moment: you can now bind your innermost VerticalStackLayouts visibility to the value of this RadioButton.

```
<VerticalStackLayout
    IsVisible="{Binding IsChecked, Source={x:Reference
    FixedRadioButton}}">
```

If you just wanted to allow the user to optionally turn a setting on in your UI, you could use a Switch control instead. I opted for the RadioButton as this will play very well with your extra assignment at the end of this chapter.

Bindings can start to look complicated quickly and this is a good example, but if you break it down, it can become much easier to follow. You are binding the IsVisible property on your VerticalStackLayout to the IsChecked property from the Source, which is a Reference to the RadioButton called FixedRadioButton.

Applying the Remaining Bindings

Let's apply the remaining bindings to your page and view model so that all fields now update your view model.

In your BoardDetailsPageViewModel class, you need to add the backing fields and properties to bind to

```
private bool isFixed = true;
private int numberOfColumns = 3;
private int numberOfRows = 2;
public bool IsFixed
{
    get => isFixed;
    set => SetProperty(ref isFixed, value);
}
public int NumberOfColumns
{
    get => numberOfColumns;
    set => SetProperty(ref numberOfColumns, value);
}
public int NumberOfRows
{
    get => numberOfRows;
    set => SetProperty(ref numberOfRows, value);
}
```

Then in your BoardDetailsPage.xaml file, you need to bind to those new properties with the bold sections below highlighting your additions.

Change the first RadioButton to be

```
<RadioButton
    Content="Fixed"
```

```
    x:Name="FixedRadioButton"
    IsChecked="{Binding IsFixed}" />
```

Then change the Entry that follows after the RadioButton to be

```
<Entry
    Text="{Binding NumberOfColumns}"
    Keyboard="Numeric" />
And finally change the Entry that follows that to be
<Entry
    Text="{Binding NumberOfRows}"
    Keyboard="Numeric" />
```

MultiBinding

There can be occasions when you wish to bind multiple source properties to a single target property in a view. To take a minor detour, let's rework your ClockWidgetViewModel to have two properties: one with the date and one with the time. You should end up with the following code (the **bold** highlights the new parts):

```
namespace WidgetBoard.ViewModels;
public class ClockWidgetViewModel : ViewModelBase
{
    private readonly Scheduler scheduler = new();
    private DateOnly date;
    private TimeOnly time;
    public ClockWidgetViewModel()
    {
        SetTime(DateTime.Now);
    }
    public DateOnly Date
    {
```

```
        get => date;
        set => SetProperty(ref date, value);
    }
    public TimeOnly Time
    {
        get => time;
        set => SetProperty(ref time, value);
    }
    private void SetTime(DateTime dateTime)
    {
        Date = DateOnly.FromDateTime(dateTime);
        Time = TimeOnly.FromDateTime(dateTime);
        scheduler.ScheduleAction(
            TimeSpan.FromSeconds(1),
            () =>
            {
                SetTime(DateTime.Now);
            });
    }
}
```

The change in the view model actually opens up a number of possibilities for you. You could

- Add separate `Labels` to render the information in different locations
- Make use of a `MultiBinding` and render both pieces of information in a single `Label`

It is the latter you will be using here. Open your `ClockWidgetView.xaml` file and make the changes you see in **bold**.

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

```
<Label
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Views.ClockWidgetView"
    FontSize="80"
    VerticalOptions="Center"
    HorizontalOptions="Center">
    <Label.Text>
        <MultiBinding StringFormat="{}{0} {1}">
            <Binding Path="Date" />
             <Binding Path="Time" />
        </MultiBinding>
    </Label.Text>
</Label>
```

To list what you have done here, you have

- Removed the `Text="{Binding Time}"` line
- Moved the above functionality into the `MultiBinding` section

You should notice a slightly different syntax to the single binding approach. In fact, you can write a single binding in a similar way, such as

```
<Label.Text>
    <Binding Path="Time" />
</Label.Text>
```

However, I am sure you can appreciate that the original `Text="{Binding Time}"` is a lot more concise and easier to read. Each of the properties that you covered under the “Binding” section applies to each of the `Binding` elements under `MultiBinding`.

You must supply either a `StringFormat` or a `Converter` in a `MultiBinding` or an exception will be thrown. The reason for this is to allow for the multiple values to be mapped down to the single value on the target.

Command

Very often you will need your applications to respond to user interaction. This can be by tapping or clicking on a button or selecting something in a list. This interaction is recorded in your view, but you usually require that the logic to handle this interaction be performed in the view model. This comes in the form of a `Command` and an optional associated `CommandParameter` set of properties. A command works in a similar way to an `event`; you can provide a method that will be executed when an event happens; commands are suited to the MVVM architecture because it enables you to bind the command to an instance in the view model, which is where you want your business logic to reside. The `Command` property itself can be bound from the view to the view model and allows the view model to not only handle the interaction but also to determine whether the interaction can be performed in the first place. You already added a `Button` to your `BoardDetailsPage.xaml` file but you didn't hook it, so let's do exactly that!

You just need to modify your button to be (changes in **bold**)

```
<Button
    Text="Save"
    HorizontalOptions="End"
    Command="{Binding SaveCommand}" />
```

Based on the binding content that you have explored, you can say that this `Buttons` `Command` property is now bound to a property on your view model called `SaveCommand`. You haven't actually created this property yet. If you are thinking it would be great if the tooling could know this and report it to me, then the next section has got you covered. "Compiled Bindings" will show you how to inform the tooling of how to report it to you. First, though, open your `BoardDetailsPageViewModel.cs` file and add your command implementation.

Your implementation comes in multiple parts.

1. You define the property itself:

   ```
   public Command SaveCommand { get; }
   ```

   You typically define a command as a read-only property as you rarely want it to change. You will likely come across commands being defined with the use of the `ICommand` interface rather than the `Command` class. The reason you are using the latter is so that you can make use of a specific method (see number 3 in this list) to update some of your views.

2. You define what action will be performed when the command is executed (basically when the `Button` is tapped/clicked in this scenario).

   ```
   public BoardDetailsPageViewModel()
   {
       SaveCommand = new Command(
           () => Save(),
           () => !string.IsNullOrWhiteSpace(BoardName));
   }
   private void Save()
   {
       var board = new Board
       {
   ```

```
            Name = BoardName,
            NumberOfColumns = NumberOfColumns,
            NumberOfRows = NumberOfRows
        };
    }
```

The Command class takes two parameters. The first is the action to perform when the command is executed, and the second, which is optional, is a way of defining whether the command can be executed. A good use case for this is if you wish to make sure that the user has entered all the required information. In your scenario, you will make sure that the user has entered a name for the board.

3. You notify the view when the status of whether the command can be executed changes. To be clear, you don't have to know that the status has changed; you can simply inform the view that it should re-query the status. This is where the Command class and its ChangeCanExecute method come in. For this, you need to tweak your BoardName property to the following:

```
public string BoardName
{
    get => boardName;
    set
    {
        SetProperty(ref boardName, value);
        SaveCommand.ChangeCanExecute();
    }
}
```

This change means that every time the BoardName property changes (and this will be done via the binding from the view), the Button that is bound to the SaveCommand will re-query to check whether the command can be executed. If it can, the Button will be enabled and the user can interact with it; if not, it will be disabled.

Compiled Bindings

Compiled bindings are a great feature that you should in almost all cases turn on! They help to speed up your applications because they help the compiler know what the bindings will be set to and reduce the amount of reflection that is required. Reflection is notoriously bad for performance so wherever possible it is highly recommended to avoid using it. Bindings by default do use an amount of reflection in order to handle the value changes between source and target. Compiled bindings, as just discussed, help to reduce this, so let's learn how to turn them on.

Compiled bindings also provide design-time validation. If you set a binding to a property on your view model that doesn't exist (imagine you made a typo, which I do a lot!), without compiled bindings, the application would still build but your binding won't do anything. With a compiled binding, the application will fail to build and the tooling will report that the property you mistyped doesn't exist.

```
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Pages.BoardDetailsPage"
    x:DataType="viewModels:BoardDetailsPageViewModel">
```

Now that you have set up your BoardDetailsPage to allow user entry and even perform an action when the Save button is interacted with, you need to structure your application so that you can see this happen.

Note that since .NET 9.0, you will see warnings reported if you do not use compiled bindings; this was implemented by the team at Microsoft in an effort to make sure developers are making the most of the performance and compile time safety that they offer.

Make Use of the BoardDetailsPage

In order to see the `BoardDetailsPage` in action, we will first need to modify the contents of the AppShell.xaml file to point to the new page. Note that we will only be tweaking this file in order to see the result of the changes we have introduced in this chapter. The next chapter will delve into much further detail on Shell.

Proceed by opening the AppShell.xaml file and modify the contents to match the following (note that the actual changes from the original content are shown in **bold**):

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

    <ShellContent
        Title="Home"
        ContentTemplate="{DataTemplate
        pages:BoardDetailsPage}" />

</Shell>
```

This will result in the new `BoardDetailsPage` being shown when we open the application.

Taking Your Application for a Spin

If you run the application, you will see that you are first presented with the screen to create a new board. You can enter the details and press Save.

Figure 5-8 shows how your application looks when it is first loaded.

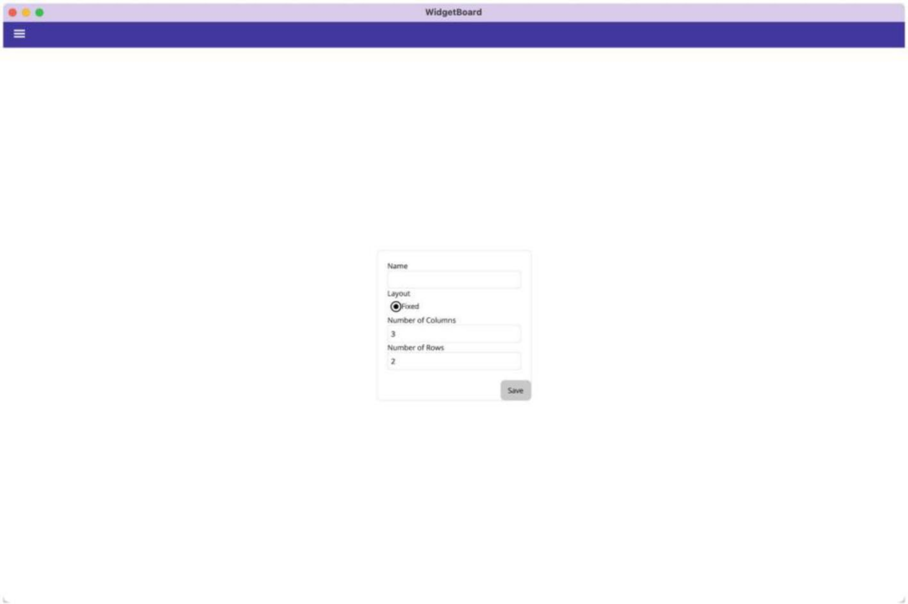

Figure 5-8. *The application home page*

It is worth noting that the Save button will not do anything just yet. Adding the handling of this button will be the topic of the next chapter when we dig deep into Shell and how to allow users to navigate around our applications.

Summary

In this chapter, you have

- Created and applied an icon for your application
- Added some placeholder pages and view models
- Filled your first page with some UI and setup bindings to the view model
- Covered data binding and its many uses
- Gained an understanding of XAML
- Learned about the possible layouts you can use to group other controls

In the next chapter, you will

- Gain an understanding of Shell and apply this to building your application's structure
- Apply the Shell navigation to allow you to navigate to your next page and the next chapter
- Make use of Shell tabs and search functionality
- Build your flyout menu using all the learnings in this chapter
- Add tabs into the application
- Add the ability to search for tabs

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch05`.

CHAPTER 6

Shell

Abstract

In this chapter, you are going to learn how to define the visual hierarchy of your .NET MAUI application and handle common concepts like navigation and search functionality – all through a concept called Shell.

Prerequisites

You need to do some setup before you can jump into using Shell. If Shell is still feeling like an unknown concept, fear not; we will be covering it in depth within this chapter.

Let's go ahead and add the following folders to your project.

Pages

Let's go ahead and create the following files under the `Pages` folder. The following steps show how to add the new pages:

- Right-click the *Pages* folder.
- Select **Add ➤ New Item**.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_6

- Select the **.NET MAUI** tab.
- Select **.NET MAUI ContentPage (XAML)**.
- Click **Add**.

BoardListPage

This is the page that will render a list of boards that users will create within your application. For now, you will not touch the contents of this file. Note that you should see `BoardListPage.xaml` and `BoardListPage.xaml.cs` files created.

You will also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method.

```
builder.Services.AddTransient<BoardListPage>();
```

FixedBoardPage

This is the page that will render the boards you create in the page created in the previous chapter. For now, you will not touch the contents of this file. Note that you should see `FixedBoardPage.xaml` and `FixedBoardPage.xaml.cs` files created.

You will also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method.

```
builder.Services.AddTransient<FixedBoardPage>();
```

SettingsPage

This is the page that will render any settings that the user can modify, for example, how frequently to refresh the widgets. For now, you will not touch the contents of this file. Note that you should see `SettingsPage.xaml` and `SettingsPage.xaml.cs` files created.

You will also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method.

```
builder.Services.AddTransient<SettingsPage>();
```

ViewModels

This houses your ViewModels that are the backing for both your Pages and Views. You created this folder in the previous chapter, but you need to add a number of classes. The following steps show how to add the new pages:

- Right-click the *ViewModels* folder.
- Select **Add ➤ New Class**.
- Click **Add**.

AppShellViewModel

This serves as the view model for the `AppShell` file that is created for you by the tooling.

```
namespace WidgetBoard.ViewModels;

public class AppShellViewModel : BaseViewModel
{
}
```

You also need to jump over to the `MauiProgram.cs` file and register this page with the Services inside the `CreateMauiApp` method.

```
builder.Services.AddTransient<AppShellViewModel>();
```

BoardListPageViewModel

This serves as the view model for the BoardListPage file that will be responsible for displaying all available boards within the application to the user.

```
namespace WidgetBoard.ViewModels;

public class BoardListPageViewModel : BaseViewModel
{
}
```

You also need to jump over to the MauiProgram.cs file and register this page with the Services inside the CreateMauiApp method.

```
builder.Services.AddTransient<BoardListPageViewModel>();
```

FixedBoardPageViewModel

This serves as the view model for the FixedBoardPage file you created.

```
namespace WidgetBoard.ViewModels;

public class FixedBoardPageViewModel : BaseViewModel
{
}
```

You also need to jump over to the MauiProgram.cs file and register this page with the Services inside the CreateMauiApp method.

```
builder.Services.AddTransient<FixedBoardPageViewModel>();
```

You should have noticed a common pattern with the creation of these files and the need to add them to the MauiProgram.cs file. This is to allow you to fully utilize the dependency injection provided by the framework, which you learned about in Chapter 3.

SettingsPageViewModel

This serves as the view model for the SettingsPage file you created.

```
namespace WidgetBoard.ViewModels;

public class SettingsPageViewModel : BaseViewModel
{
}
```

You also need to jump over to the MauiProgram.cs file and register this page with the Services inside the CreateMauiApp method.

```
builder.Services.AddTransient<SettingsPageViewModel>();
```

With that concluding the prerequisites required for this chapter, let's proceed onto learning all about Shell and how we can define the structure of .NET MAUI applications.

Shell

Shell in .NET MAUI enables you to define how your application will be laid out, not in terms of actual visuals but by defining things like whether you want your pages viewed in tabs or just a single page at a time. It also enables you to define a flyout, which is a side menu in your application. You can choose to have it always visible or toggle it to slide in/out, and this can also vary based on the type of device you are running on. Typically a desktop has more visual real estate, so you may wish to keep the flyout always open then.

For your application, you are going to make use of the flyout to allow you to define multiple boards that you can configure and load. I really like the idea of having one board for when I work and then swapping to something else when working on a side project or even for gaming.

To save having to return to this area and change bits, you are going to jump straight into the more in-depth option and feature-rich outcome. Don't worry, though; as you discover each new concept, you will dive into some detail to cover what it is and why you are using it along with then applying that concept to your application.

ShellContent

If you take a look at your AppShell.xaml file, you should see very little inside. Currently it has the following line:

```
<ShellContent
    ContentTemplate="{DataTemplate pages:BoardDetailsPage}" />
```

You will recall that in the previous chapter we modified the contents to the above in order to show our progress when running the application. We didn't dig into the details of the change in order to keep that detail within the Shell chapter, so let's explore what it means.

Your application's main content will now be an instance of your recently created BoardDetailsPage. You don't need the Title or Route options anymore as you will be controlling them in different ways.

The Title property will be set based on the page that is shown, so you will learn about this a little later on.

The Route property you will control as part of the next section, "Navigation."

Finally, you added xmlns:pages="clr-namespace:WidgetBoard.Pages" to the top of the file in order to be able to refer to the BoardDetailsPage.

Navigation

I am personally a fan of simplifying the code I write so long as it continues to make it easy to read. With this in mind, I would like to suggest you improve on the registration of your pages and their view models already.

Registering Pages for Navigation

Therefore, I suggest that you create a new method into your `MauiProgram.cs` file.

```
private static void AddPage<TPage, TViewModel>(
    IServiceCollection services,
    string route)
    where TPage : Page
    where TViewModel : BaseViewModel
{
    services
        .AddTransient(typeof(TPage))
        .AddTransient(typeof(TViewModel));
    Routing.RegisterRoute(route, typeof(TPage));
}
```

Notice the line `Routing.RegisterRoute(route, typeof(TPage));`. This serves as a very important part in this topic of navigation. It means that when you tell Shell to navigate to a specific route, it will create a new instance of the `TPage` type you passed in and navigate to it. Of course, because you have registered these types with the dependency injection layer, it means that any dependencies that are defined as parameters to the constructor will be created and passed in for you.

The above then means that rather than writing

```
services.AddTransient<BoardDetailsPage>()
services.AddTransient<BoardDetailsPageViewModel>()
Routing.RegisterRoute(route, typeof(TPage));
```

you can now write

```
AddPage<BoardDetailsPage, BoardDetailsPageViewModel>(builder.
Services, "boarddetails");
```

with the added change that you now define this `route`. So let's go and delete your old registrations and replace with

```
AddPage<BoardDetailsPage, BoardDetailsPageViewModel>(
    builder.Services, RouteNames.BoardDetails);
  AddPage<BoardListPage, BoardListPageViewModel>(
    builder.Services, RouteNames.BoardList);
  AddPage<FixedBoardPage, FixedBoardPageViewModel>(
    builder.Services, RouteNames.FixedBoard);
  AddPage<SettingsPage, SettingsPageViewModel>(
    builder.Services, RouteNames.Settings);
```

I also recommend defining the routes as constant strings somewhere in your code base to avoid typos when wanting to navigate to them – which is why the last parameter in all of the calls you just added refers to something called `RouteNames`; this class does not exist, so let's create this now.

Add a new class file and call it *RouteNames.cs* and then modify the contents to the following:

```
namespace WidgetBoard;

public static class RouteNames
{
    public const string BoardDetails = "boarddetails";
```

```
    public const string BoardList = "boards";
    public const string FixedBoard = "fixedboard";
    public const string Settings = "settings";
}
```

This means you can save one line of code per page and view model pair that you had registered as well as the code to register the route for navigation. The added benefit of introducing the RouteNames class means that you reduce the risk of a typo being introduced because the string only needs to be defined in a single place. In fact, this means that even if there is a typo the code will still likely work because the typo will apply everywhere it is used within the app.

Note that as a further enhancement, if you are making use of the .NET MAUI Community Toolkit – which I would thoroughly recommend you do. You can make use of the AddTransientWithShellRoute method; this would remove the need to write your own AddPage method that we did in this section and therefore give you less code to maintain.

Now that you have registered your pages, let's take a look at how you can actually perform navigation.

Performing Navigation

There are multiple ways to specify the route for navigation, but they all use the Shell.Current.GoToAsync method.

So, for example, you could navigate to your FixedBoardPage with the following:

```
await Shell.Current.GoToAsync(RouteNames.FixedBoard);
```

This will result in a `FixedBoardPage` being created and pushed onto the navigation stack. This is precisely the behavior that you need at the end of your `SaveCommand` execution in your `BoardDetailsPagesViewModel` class.

Navigating Backward

You can also pop pages off the navigation stack by navigating backward. This can be achieved by the following:

```
await Shell.Current.GoToAsync("..");
```

with the .. component telling Shell that it needs to go backward. In fact, backward and forward navigation can be performed together:

```
await Shell.Current.GoToAsync($"../{RouteNames.BoardList}");
```

Passing Data When Navigating

One key thing that you really need to do as part of creating your board and navigating to the page that will render the board is to pass the context across to that page so it knows what to render. There are multiple ways to both send the data and also to receive it.

Let's start with sending.

- You can pass primitive data through the query string itself, for example:

  ```
  await Shell.Current.GoToAsync($"{RouteNames.
  FixedBoard}?boardid=1234");
  ```

  By providing the `boardid`, you put the responsibility on the receiving page (or page view model) to retrieve the right board by using the specified ID.

- More complex data can be sent as an `IDictionary<string, object>` parameter in the `GoToAsync` method, such as

```
await Shell.Current.GoToAsync(
    RouteNames.FixedBoard,
    new Dictionary<string, object>
    {
        { "Board", board }
    });
```

You can also send a complex object like the above, which means the originating page (or page view model) is responsible for retrieving or constructing the board and you send the whole thing to the receiving page.

There are two main ways to handle sending complex data when navigating with Shell. Let's take a look at each in turn.

IQueryAttributable

To receive data, you can implement the `IQueryAttributable` interface provided with .NET MAUI. Shell will either call this on the page you are navigating to, or if the `BindingContext` (your view model) implements the interface, it will call it there. Add this to your `FixedBoardPageViewModel` class because you are going to need to process the data. You will be going with the complex object option because you have already loaded the Board in your `AppShellViewModel` class.

```
public void ApplyQueryAttributes(IDictionary<string,
object> query)
{
    var board = query["Board"] as FixedBoard;
}
```

You aren't going to do anything with this data just yet, but it is ready for when you start to build your board layout view in the next chapter. For now, you will continue on with the theme of Shell and define your flyout menu.

You will also need to make your `FixedBoardPageViewModel` implement the `IQueryAttributable` interface. Change the class definition from

```
public class FixedBoardPageViewModel : BaseViewModel
```

to the following (changes in **bold**):

```
public class FixedBoardPageViewModel : BaseViewModel,
IQueryAttributable
```

Note that you will also need to add the following using statement to the top of your *FixedBoardPageViewModel.cs* file:

```
using WidgetBoard.Models;
```

QueryProperty

An alternative to using the `IQueryAttributable` interface is to make use of the `QueryProperty` in your receiving class. Making use of the same example from the "Passing Data When Navigating" section, we could (but we won't so don't worry to apply any of these changes) change the `FixedBoardPageViewModel` class to the following:

```
using WidgetBoard.Models;

namespace WidgetBoard.ViewModels;

[QueryProperty(nameof(CurrentBoard), "Board")]
public class FixedBoardPageViewModel : BaseViewModel
{
    public Board CurrentBoard { get; set; }
}
```

You can see from the changes above in **bold** that we have created a property called `CurrentBoard` and then added the `QueryProperty` attribute to the class. This attribute instructs Shell to set the `CurrentBoard` property (first parameter) when a value is received in the query string with the key of “`Board`”.

The main reason why I prefer `IQueryAttributable` over `QueryProperty` is that .NET MAUI will call the method for us during navigation; if we wanted to handle the navigation in our view model without this interface implementation, we would have to add additional boilerplate code to do so.

Let’s proceed to learning about the next Shell feature in order to connect all the dots and have a working application with navigation by the end of this chapter.

Flyout

A flyout is a menu for a Shell application that is accessible through an icon or by swiping from the side of the screen. The flyout can consist of an optional header, flyout items, optional menu items, and an optional footer.

For your application, you are going to provide a basic header, and then the main content will be a dynamic list of all the boards your user creates. This means that you are going to have to override the main content, but thankfully Shell makes this an easy task.

The first thing I like to do when working on a new XAML file is to turn on compiled bindings, which I covered earlier. If you recall, this is by specifying the `x:DataType` attribute to tell the compiler the type that your view will be binding to. Let’s do that now; first, open up the *AppShell.xaml* file and make the following changes (in **bold**):

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    x:Class="WidgetBoard.AppShell"
```

```
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:DataType="viewmodels:AppShellViewModel"
Shell.FlyoutBehavior="Flyout">
```

This helps you as you build the view to see what doesn't exist in your view model. Of course, if you prefer to build the view model first, then this also helps.

Finally, you need to add `xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"` to the top of the file.

Now we want to proceed to defining how our Flyout menu will be presented. This can be customized with the `FlyoutHeader` and `FlyoutContent`, so let's take a look at each one in turn.

FlyoutHeader

The `FlyoutHeader` can be given any control or layout, and therefore, you can build a really good-looking header option. For your application, you are just going to add a title Label.

Below your `ShellContent` element, you want to add the following:

```
<Shell.FlyoutHeader>
    <Label
        Text="My boards"
        FontSize="20"
        HorizontalTextAlignment="Center" />
</Shell.FlyoutHeader>
```

Hopefully the above is self-explanatory, but to cover the parts I haven't already covered, you have the ability to specify different layout information in a Label so you can make the text centered. It is usually recommended that you use the `HorizontalOptions` property over the `HorizontalTextAlignment` property for performance reasons; however, if you try that here, you will see that it doesn't center the `Label`.

Now let's add in the main part of your menu.

FlyoutContent

First, if you want to use a static set of items in your menu, you can simply add `FlyoutItems` to the content. This can work well when you have a fixed set of pages such as Settings, Home, and so on. You will be showing the boards that the user creates, so you will need something dynamic. For this, you need to supply the `FlyoutContent`. More importantly, it's your first introduction to the `CollectionView` control.

The `CollectionView` allows you to define how an item will look and then have it repeated for each item in a collection that is bound to it. Additionally, the `CollectionView` provides the ability to allow the user to select items in the collection, and you can define behavior that will be performed when that selection happens. Let's add the following to your Shell:

```
<Shell.FlyoutContent>
    <CollectionView
        ItemsSource="{Binding Boards}"
        SelectionMode="Single"
        SelectedItem="{Binding CurrentBoard}">
        <CollectionView.ItemTemplate>
            <DataTemplate x:DataType="models:Board">
                <Label
                    Text="{Binding Name}"
                    FontSize="20"
                    Padding="10,0,0,0" />
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</Shell.FlyoutContent>
```

You also need to add xmlns:models="clr-namespace:WidgetBoard.Models" to the top of the file.

If we deconstruct the XAML that was just added, we can make the following statements. Your `FlyoutContent` will display a collection of items; each item will be presented as a `Label` set to the `Name` of each item. The items will be `Board` instances in the collection of `Boards` in your view model. Additionally, the `CurrentBoard` property on your view model will be updated when the user selects one of the `Labels` in this collection.

If you have added all of the parts I have discussed, you will likely notice that the tooling is reporting that you haven't added the `Boards` or `CurrentBoard` properties that you are binding to over in your view model. Let's jump over to your `AppShellViewModel.cs` file and add the following.

Collection of Boards

```
public ObservableCollection<Board> Boards { get; } = [];
```

The `ObservableCollection` is a special type of collection that implements `INotifyCollectionChanged`. This means that anything bound to it will monitor changes to the collection and update its contents on screen. Note that the use of `ObservableCollection` above means that the UI will only respond to changes inside the collection; if you were to assign a new value to the `Boards` property, then this would not update the UI unless you implement `INotifyPropertyChanged` and raise the `PropertyChanged` event.

Additionally, for now, you will add a fixed entry into this `Boards` collection to make it possible to interact with. Later you will be saving to and loading from a database.

```
public AppShellViewModel()
{
    Boards.Add(
        new Board
        {
            Name = "My first board",
```

```
            NumberOfColumns = 3,
            NumberOfRows = 2
        });
}
```

Note that you will also need to add the following using statements to the top of your file:

```
using System.Collections.ObjectModel;
using WidgetBoard.Models;
```

Selected Board

You bound the SelectedItem property from the CollectionView to your CurrentBoard property. When your property changes, you can navigate to the board that was selected.

```
private Board? currentBoard;
public Board? CurrentBoard
{
    get => currentBoard;
    set
    {
        if (SetProperty(ref currentBoard, value) &&
            value is not null)
        {
            BoardSelected(value);
        }
    }
}
```

You may recall that I discussed in Chapter 4 the potential value of SetProperty returning a Boolean value. You have finally found a use for it! You only want to handle a board selection change if the CurrentBoard property really has changed.

Navigation to the Selected Board

Following on from the “Navigation” section earlier, you will navigate to the route “fixedboard” which your `FixedBoardPage` is configured to. You will also pass in the selected board so that it can be presented on screen.

```
private async void BoardSelected(Board board)
{
    await Shell.Current.GoToAsync(
        RouteNames.FixedBoard,
        new Dictionary<string, object>
        {
            { "Board", board }
        });
}
```

Before your bindings will work, you need to make some further changes.

Setting the BindingContext of Your AppShell

Let’s change the constructor of your `AppShell.xaml.cs` file to set the `BindingContext`.

```
public AppShell(AppShellViewModel appShellViewModel)
{
    InitializeComponent();
    BindingContext = appShellViewModel;
}
```

Note that you will also need to add the following `using` statement to the top of your file:

```
using WidgetBoard.ViewModels;
```

You should recall that you added the AppShellViewModel as a transient in the MauiProgram.cs file, meaning that you will be provided with a new instance when your AppShell class is created for you. You will also need to do the same for the AppShell class because we have given it a dependency now.

Register AppShell with the MAUI App Builder

Let's register AppShell in your MauiProgram.cs file.

```
builder.Services.AddTransient<AppShell>();
```

Resolve the AppShell Instead of Creating It

Change the contents of your App.xaml.cs file to be as follows (changes in **bold**):

```
namespace WidgetBoard;

public partial class App : Application
{
    private readonly AppShell appShell;

    public App(AppShell appShell)
    {
        this.appShell = appShell;
        InitializeComponent();
    }

    protected override Window CreateWindow(IActivationState?
    activationState)
    {
        return new Window(this.appShell);
    }
}
```

All of the above changes allow you to use `AppShell` just like any other page and not have to create an instance manually.

Taking Your Application for a Spin

If you run the application, you will see that you are first presented with the screen to create a new board. You can enter the details and press Save.

Figure 6-1 shows how your application looks when it is first loaded.

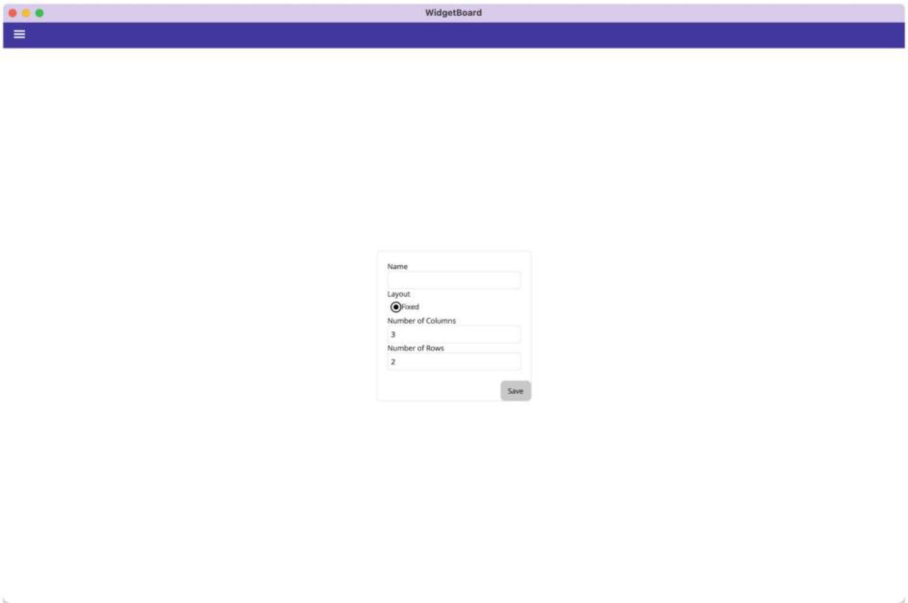

Figure 6-1. *The application home page*

Or you can slide out the menu from the left-hand side. Figure 6-2 shows the flyout menu in your application.

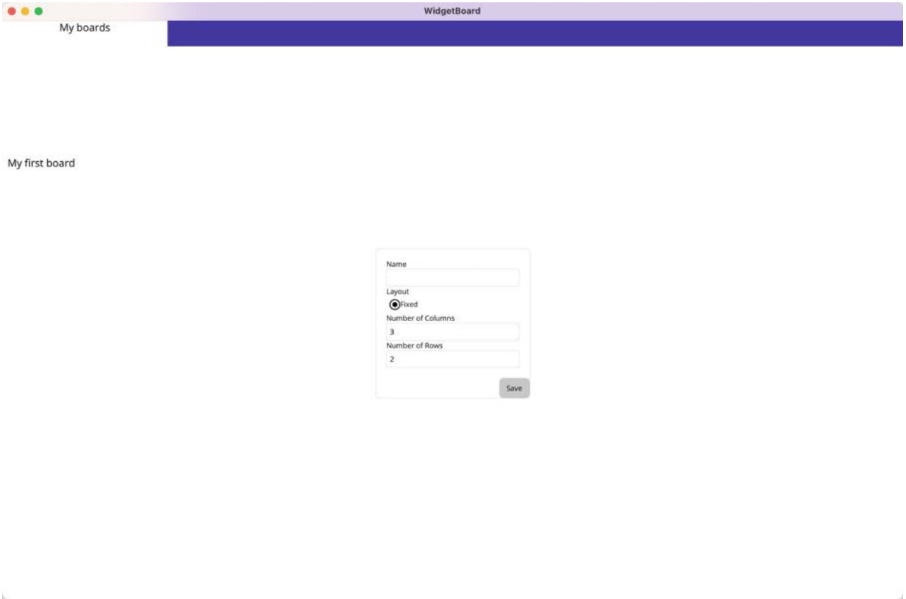

Figure 6-2. *The application flyout menu*

By either selecting the board or pressing Save, you will be navigated to your `FixedBoardPage`. Figure 6-3 shows your `FixedBoardPage` displaying the default content. This is because you haven't wired up the board object that you are receiving, but it proves that your navigation and Shell setup is working.

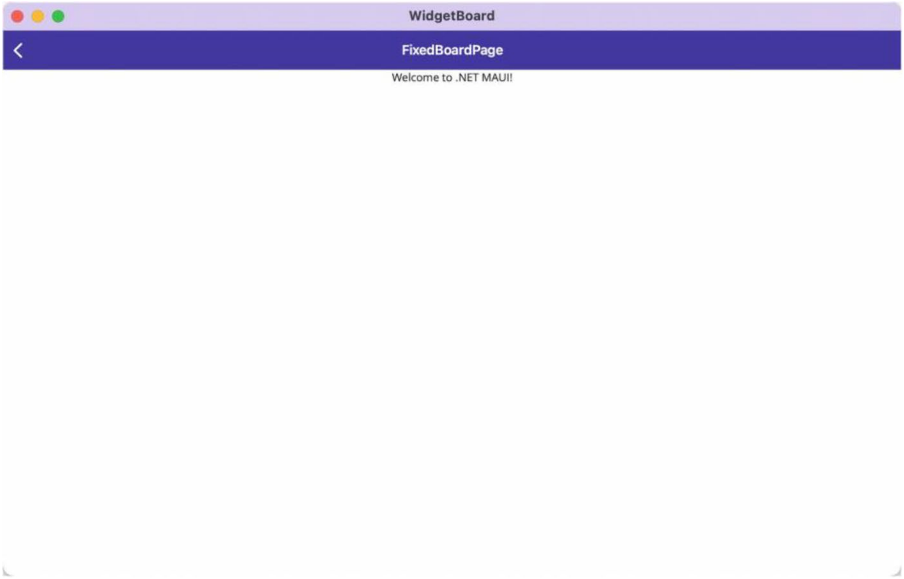

Figure 6-3. *The fixed board page after navigating*

Tabs

Shell offers many different ways to build the structure of your application; if a Flyout menu doesn't fit your application, then you might opt to use tabs instead, or even in combination. We are going to do the latter to show how you can also make use of tabs.

You will have noticed that when the application was first run, we saw the `BoardDetailsPage` which lets a user create a new board. While this might be useful on the first ever use of the application, it is not likely to be a common place where a user will want to land in our application. For this, we are going to make two key changes: introduce tabs and change the landing page for our users.

Let's first open up the AppShell.xaml file and make the following changes

We can replace these lines

```
<ShellContent
    ContentTemplate="{DataTemplate pages:BoardDetailsPage}" />
```

with the following:

```
<TabBar>
    <Tab Title="Boards">
        <ShellContent ContentTemplate="{DataTemplate pages:
        BoardListPage}" />
    </Tab>

    <Tab Title="Settings">
        <ShellContent ContentTemplate="{DataTemplate pages:
        SettingsPage}" />
    </Tab>
</TabBar>
```

This now means that we will see a tab bar at the bottom of the application; the first tab is labelled Boards and will present the `BoardListPage`, and the second tab will be called Settings and present the `SettingsPage`. We won't add any content to the `SettingsPage` yet as that will be the subject of future chapters; we added two tabs now to highlight that Shell will only present the tab bar if there is more than one tab within the bar.

We will now apply the same approach that was added to displaying the user's boards in the flyout menu. First, let's open up the `BoardListPage.xaml` file and make the following changes.

Modify the ContentPage element to look as follows (changes in **bold**):

```
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
```

```
    xmlns:models="clr-namespace:WidgetBoard.Models"
    x:Class="WidgetBoard.Pages.BoardListPage"
    x:DataType="viewModels:BoardListPageViewModel"
    Title="My boards">
```

Inside the ContentPage element, add

```
<CollectionView
ItemsSource="{Binding Boards}"
SelectionMode="Single"
SelectedItem="{Binding CurrentBoard}">
<CollectionView.ItemTemplate>
    <DataTemplate x:DataType="models:Board">
        <Label
            Text="{Binding Name}"
            FontSize="20"
            Padding="10,0,0,0" />
    </DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
```

This is the same as we added to *AppShell.xaml* so we won't cover what this does again.

Now let's open the *BoardListPageViewModel.cs* file and make the following addition.

Collection of Boards

Add a collection of boards and populate it.

```
public ObservableCollection<Board> Boards { get; } = [];

public BoardListPageViewModel()
{
    Boards.Add(
```

```
        new Board
        {
            Name = "My first board",
            NumberOfColumns = 3,
            NumberOfRows = 2
        });
}
```

Note that you will also need to add the following `using` statements to the top of your file:

```
using System.Collections.ObjectModel;
using WidgetBoard.Models;
```

Selected Board

You bound the `SelectedItem` property from the `CollectionView` to your `CurrentBoard` property. When your property changes, you can navigate to the board that was selected.

```
private Board? currentBoard;
public Board? CurrentBoard
{
    get => currentBoard;
    set
    {
        if (SetProperty(ref currentBoard, value) &&
            value is not null)
        {
            BoardSelected(value);
        }
    }
}
```

Navigation to the Selected Board

Following on from the “Navigation” section earlier, you will navigate to the route “fixedboard” which your `FixedBoardPage` is configured to. You will also pass in the selected board so that it can be presented on screen.

```
private async void BoardSelected(Board board)
{
    await Shell.Current.GoToAsync(
        RouteNames.FixedBoard,
        new Dictionary<string, object>
        {
            { "Board", board}
        });
}
```

Before your bindings will work, you need to make some further changes.

Setting the BindingContext of Your BoardListPage

Let’s change the constructor of your `BoardListPage.xaml.cs` file to set the `BindingContext`.

```
public BoardListPage(BoardListPageViewModel
boardListPageViewModel)
{
    InitializeComponent();
    BindingContext = boardListPageViewModel;
}
```

Note that you will also need to add the following `using` statement to the top of your file:

```
using WidgetBoard.ViewModels;
```

This concludes the changes to add a set of tabs to our application. Let's have a look at how it presents.

Taking Your Application for a Spin

If you run the application, you will see that you are presented with two tabs at the bottom of the application. The main content will present the fixed list of "My first board". Figure 6-4 shows the application presenting a list of boards and the option to switch tabs at the bottom. You can click/tap on the "My first board" item, which will navigate to the page for that board.

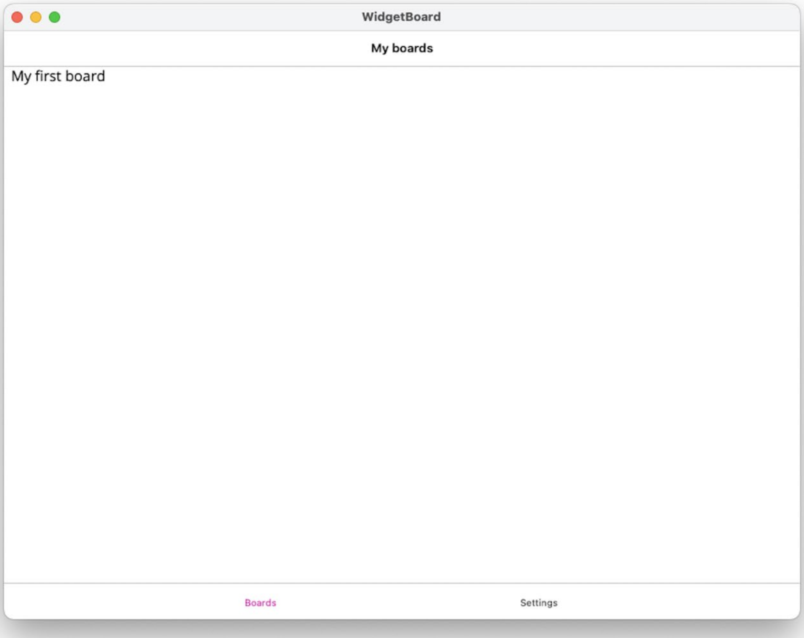

Figure 6-4. *The application with tabs*

One final feature to cover in this chapter is the ability to provide the user the ability to search for boards.

Search

Shell allows you to create your own `SearchHandler`, which means you can define how the results are met with the values entered in the search box that is automatically provided. If we imagine that the user has created a lot of boards, they will need a quick way to find the board that they wish to display.

Let's give the users the ability to search for their boards in the application. First, we need to create our `SearchHandler` implementation. Let's do this by adding a new class to the root of the project.

- Right-click the *WidgetBoard* project.
- Select **Add ➤ New Class**.
- Enter the name **BoardSearchHandler**.
- Click **Add**.

Now let's incrementally add our changes to this file.

Add Our Data

Again we are faking the data until we reach a later chapter, but let's add the following into the class:

```
private readonly IList<Board> boards =
[
    new Board
    {
        Name = "My first board"
    },
    new Board
    {
        Name = "My second board"
    },
```

```
    new Board
    {
        Name = "My third board",
    }
];
```

This gives us a list of three boards that we will be able to search against.

Inherit from SearchHandler

The next step is to make our `BoardSearchHandler` class inherit from `SearchHandler`. Let's make the following **bold** change:

```
public class BoardSearchHandler : SearchHandler
```

Handling the OnQueryChanged Method

The `OnQueryChanged` allows us to provide the search results back to Shell so that it can present them to the user. To do this, we `override` the method as follows:

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

    if (string.IsNullOrWhiteSpace(newValue))
    {
        ItemsSource = null;
    }
    else
    {
        ItemsSource = boards
```

```
            .Where(board => board.Name.Contains(newValue,
            StringComparison.CurrentCultureIgnoreCase))
            .ToList<Board>();
    }
}
```

The method accepts an `oldValue` and a `newValue` parameter; for our scenario, we only care about the `newValue` parameter. We will check whether the user has entered anything; if they haven't, we set the `ItemsSource` to null, meaning that Shell will hide any results. If the user has entered a value, then we check our `boards` field and whether any of the names contains the entered text; we then assign the results to the `ItemsSource` property so that Shell can present them.

Handling the OnItemSelected Method

The final change in this class is to `override` the `OnItemSelected` method to handle when the user selects a result.

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

    // Let the animation complete
    await Task.Delay(1000);

    await Shell.Current.GoToAsync(
        RouteNames.FixedBoard,
        new Dictionary<string, object>
        {
            { "Board", (Board)item}
        });
}
```

We will wait for one second to allow for the Shell navigation to finish before we then navigate to the `FixedBoardPage`. The navigation code should look very similar to the other navigation code that we added throughout this chapter.

Using the BoardSearchHandler

The final change in this section is to add the newly created `BoardSearchHandler` class into the *BoardListPage.xaml* file in order to instruct Shell on how to provide search functionality to the user.

Open the *BoardListPage.xaml* file and add the following code above the CollectionView entry:

```
<Shell.SearchHandler>
    <widgetBoard:BoardSearchHandler
        Placeholder="Enter board name"
        ShowsResults="True">
        <SearchHandler.ItemTemplate>
            <DataTemplate x:DataType="models:Board">
              <Label
                  Text="{Binding Name}"
                  FontSize="20"
                  Padding="10,0,0,0" />
            </DataTemplate>
        </SearchHandler.ItemTemplate>
    </widgetBoard:BoardSearchHandler>
</Shell.SearchHandler>
```

The first two properties that we are setting should be self-explanatory; the third may not – because the `BoardSearchHandler` class is setting the `ItemsSource` property to a list of `Board` instances, we need to tell Shell which property on the `Board` class should be used to present text to the user. The `ItemTemplate` property provides a template that will be created and displayed for each search result returned.

Note that you will also need to add the namespace `xmlns:widgetBoard="clr-namespace:WidgetBoard"` into the `ContentPage` element.

This concludes the changes for searching and also this chapter; let's proceed to running the application for a final time.

Taking Your Application for a Spin

If you run the application, you will see that there is a search box in the title bar with the placeholder of "Enter board name". Figure 6-5 shows the application with the search box in the title bar.

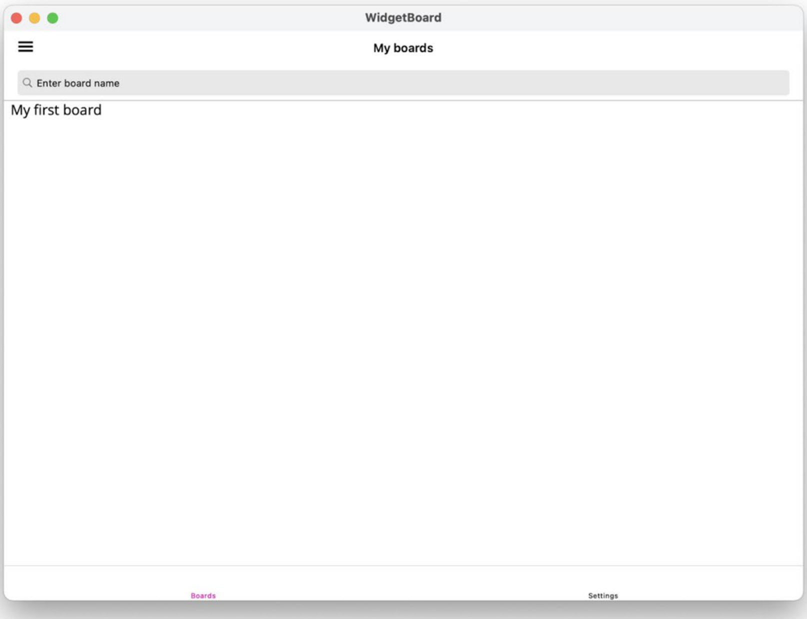

***Figure 6-5.** The application showing the search box*

The user will be presented with search results based on the text entered within the search box. Figure 6-6 shows the application matching the entered text of “ir” to both “My first board” and “My third board”.

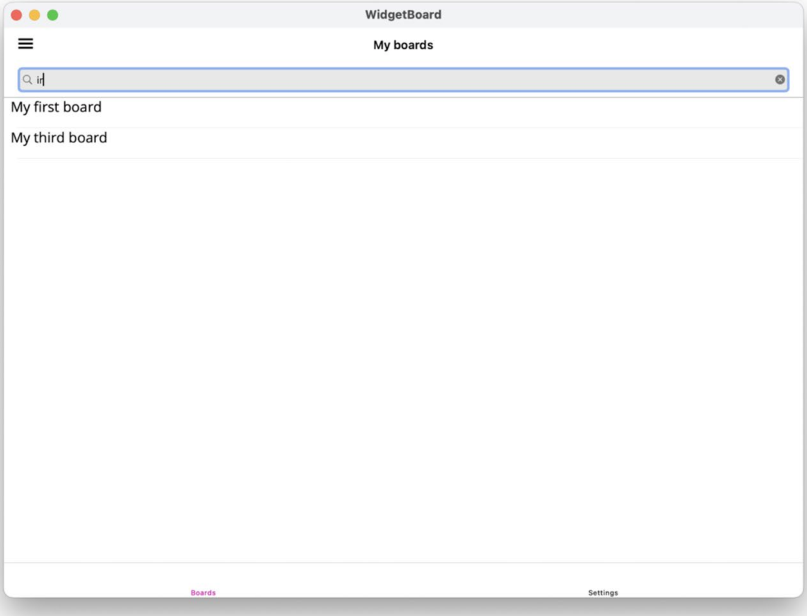

Figure 6-6. *The application showing the search box with search results*

ToolbarItems

This feature might not strictly belong to Shell, but it fits into the shell of the application. We have already added a search bar to the title bar of our application; we can also add buttons to that bar in order to provide the user with quick ways of achieving tasks. We have the perfect scenario in our application – there is currently no way to add a board, which will become a little frustrating for users if we don’t fix that.

The changes in this section will actually teach us three new concepts: how to add buttons onto the title bar, how to present a page without navigating to it, and how to show a page and wait for a result to be returned. Let's proceed to doing this.

Add a ToolbarItem to a ContentPage's ToolbarItems

The `ContentPage` class provides us with the ToolbarItems property; this makes it possible for all pages in an application to define buttons on the title bar and assign specific actions that can be performed based on when they are interacted with.

In order to introduce an add button, you need to open the *BoardListPage.xaml* file and add the following code above the `<Shell.SearchHandler>` element:

```
<ContentPage.ToolbarItems>
    <ToolbarItem Text="Add" Command="{Binding AddBoardCommand}" />
</ContentPage.ToolbarItems>
```

You can see that we have added a single `ToolbarItem` into the `ToolbarItems` collection. Our item has the `Text` of Add; you could also add an image icon if you wanted to. Finally, we set the `Command` property by binding it to a property called `AddBoardCommand` on the view model behind this page. Based on that last part, we now need to add that property to the view model; let's do that now.

Open up the *BoardListPageViewModel.cs* file and make the following changes.

Introduce the `AddBoardCommand` property.

```
public ICommand AddBoardCommand { get; }
```

Initialize the `AddBoardCommand` property; inside the constructor, add the following line:

```
AddBoardCommand = new Command(OnAddBoard);
```

This means that the `OnAddBoard` method will be executed when the button is interacted with.

Add the method that will be executed when the `AddBoardCommand` is executed.

```
private async void OnAddBoard()
{
    await Shell.Current.GoToAsync(RouteNames.BoardDetails);
}
```

This doesn't do anything new just yet; it will navigate the user to the `BoardDetailsPage` because we registered the BoardDetails route to that page in our *MauiProgram.cs* file earlier on in the book. One thing I would like to highlight is that we are awaiting the call to `GoToAsync`; this means that the application will only wait for the page to be navigated to and then continue executing. This behavior is not quite what we want – we want to show a page and have it return the board that was created so we can add it to the `Boards` property and have it presented to the user. This leads us onto the next new concept.

Changing the PresentationMode of a ContentPage

By default the `GoToAsync` method in Shell will result in the new `ContentPage` that the application goes to will be navigated to – this means it will be added to the navigation stack and a back button added to the title bar to allow the user to navigate back. This behavior doesn't really fit with our scenario of showing a page to create something and then have it close. Shell provides us with the `PresentationMode` property for just these types of scenarios.

Let's proceed to making use of this `PresentationMode` property and customize our *BoardDetailsPage.xaml* file. Open the file and make the following changes.

Add the `PresentationMode` property.

Modify the ContentPage element to look as follows (changes in **bold**):

```
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Pages.BoardDetailsPage"
    Shell.PresentationMode="ModalAnimated"
    x:DataType="viewModels:BoardDetailsPageViewModel">
```

The use of the `ModalAnimated` property means that when Shell presents the `BoardDetailsPage`, it will present the page on top of the current page outside of the navigation stack. This means the title bar will be hidden; this means that currently the user will not have a way to leave our page, so let's also add in a cancel button.

Add in a cancel button.

You will want to replace the current save button code

```
<Button
    Text="Save"
    HorizontalOptions="End"
    Command="{Binding SaveCommand}" />
```

with a grid that contains a cancel and save button.

```
<Grid ColumnDefinitions="*,*,*">
    <Button
        Text="Cancel"
        Command="{Binding CancelCommand}" />

    <Button
        Text="Save"
        Grid.Column="2"
        Command="{Binding SaveCommand}" />
</Grid>
```

The above layout means that we will have three equal spaced columns in our `Grid` with the cancel `Button` filling the first column and the save `Button` filling the third column. We won't add in the `CancelCommand` to the view model just yet; we will save it for the next section.

Show a page and wait for a result.

This can be a common scenario in applications with multiple screens, and in our application, it is perfect! We want to show a page to allow a user to create a new board and then return to the screen that shows the list of boards.

The previous sections all led up to this point! We have two final changes to make in order to complete the ability to add a new board into the application.

Wait for a result to be returned from a ContentPage.

In order to do this, you need to open the *BoardListPageViewModel.cs* file that you modified earlier and update the `OnAddBoard` method to the following (with changes in **bold**):

```
private async void OnAddBoard()
{
    TaskCompletionSource<Board?> boardCreated = new();
    await Shell.Current.GoToAsync(
        RouteNames.BoardDetails,
        new Dictionary
        {
            { "Created", boardCreated }
        });

    var newBoard = await boardCreated.Task;

    if (newBoard is not null)
    {
        Boards.Add(newBoard);
    }
    await Shell.Current.GoToAsync(RouteNames.BoardDetails);
}
```

This is our first use of the `TaskCompletionSource` class; if it is your first introduction to it, let me provide some context. This class allows a developer to perform an asynchronous operation and when it has completed return a result back to the part of the application which was waiting for it. So to explain our scenario:

1. We declare a `TaskCompletionSource<Board?>`, meaning that we expect to receive a return result of type `Board` and the ? means that it could be null.

2. We pass the `boardCreated` variable as a parameter when requesting that Shell goes to the `RouteNames.BoardDetails` section of the application.

3. We `await` the result of `boardCreated` and act on the result if it is not null.

This concludes the first change; now let's proceed to providing a result back from the BoardDetailsPage.

Returning a Result from a ContentPage

The final change that we need to make is inside the *BoardDetailsPageViewModel.cs* file, so let's open it and make the following changes.

Add the following properties:

```
public ICommand CancelCommand { get; }
public TaskCompletionSource<Board?>?
BoardCreatedCompletionSource { get; set; }
```

Add a QueryProperty attribute to the class to point Shell to the property to set during navigation. Changes are in **bold**:

```
[QueryProperty(nameof(BoardCreatedCompletionSource), "Created")]
public class BoardDetailsPageViewModel : BaseViewModel
```

The above means that when a parameter named Created is provided during a Shell.Current.GoToAsync call, the property named BoardCreatedCompletionSource will be populated with the value.

Next you need to assign the CancelCommand property in the constructor as follows (changes in **bold**):

```
public BoardDetailsPageViewModel()
{
    CancelCommand = new Command(
        async () =>
        {
            await Shell.Current.GoToAsync("..");

            BoardCreatedCompletionSource?.SetResult(null);
        });
```

```
    SaveCommand = new Command(
        () => Save(),
        () => !string.IsNullOrWhiteSpace(BoardName));
}
```

The standout change here is the line `BoardCreatedCompletionSource?.SetResult(null);` this means that when the cancel button is actioned, the `OnAddBoard` method will be returned a `null` result to indicate that no boards was created.

The final change in this section and in fact this chapter is to set the result in the Save method (changes in **bold**):

```
private void Save()
{
    var board = new Board
    {
        Name = BoardName,
        NumberOfColumns = NumberOfColumns,
        NumberOfRows = NumberOfRows
    };

    Shell.Current.GoToAsync("..");

    BoardCreatedCompletionSource?.SetResult(board);
}
```

The first new line means that the current page will be hidden, and the second line means that the newly created `board` will be returned to the `OnAddBoard` method.

Taking Your Application for a Spin

This now concludes how to show a `ContentPage` in a .NET MAUI application and wait for it to return a result. Let's take the application for a final spin and observe the functionality that was just introduced. Figure 6-7 shows the application with the Add button in the title bar.

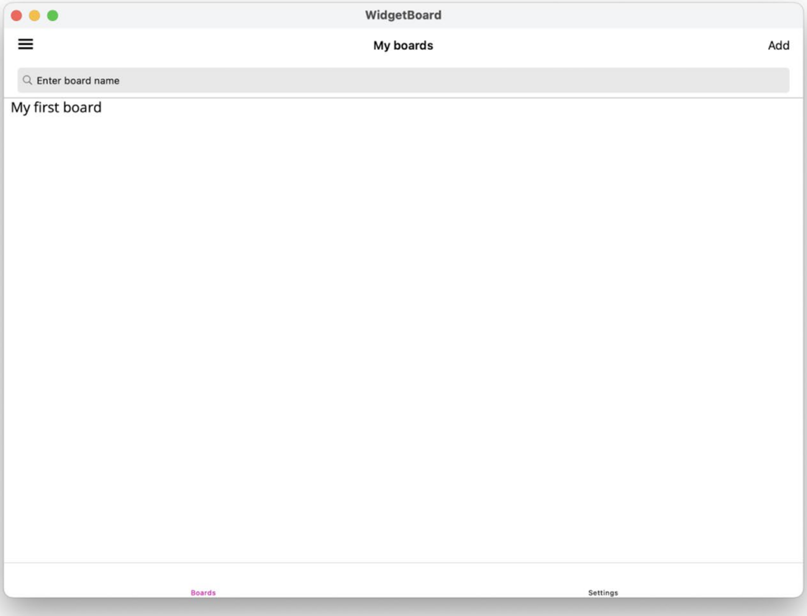

Figure 6-7. *The application showing the add button*

When you click the Add button, the application will present the `BoardDetailsPage`. Figure 6-8 shows the application presenting the ability to create a board by supplying a name, number of columns, and number of rows.

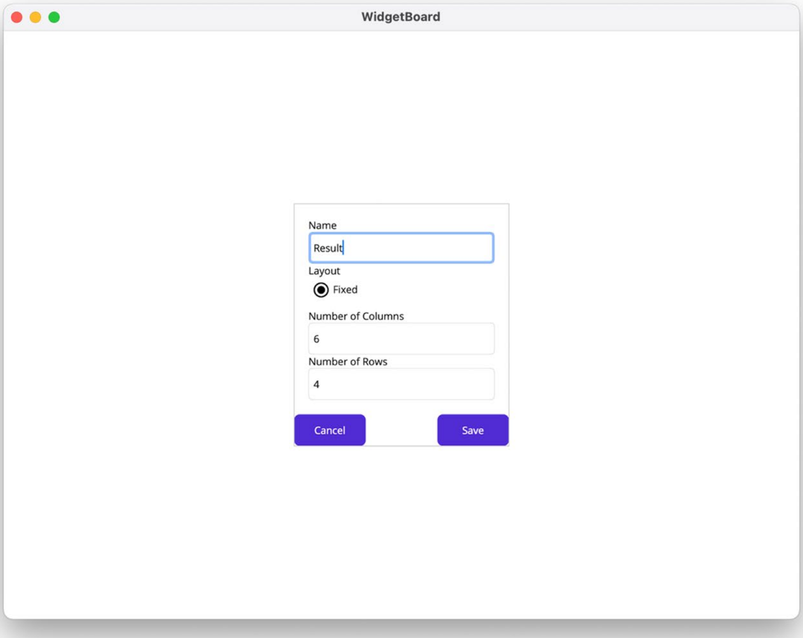

Figure 6-8. *The application showing the create a board page*

Finally, when the Save button is pressed, the user is then returned to the list of boards with the new board added to the list. Figure 6-9 shows the application presenting a list of boards including the new board named “Result”.

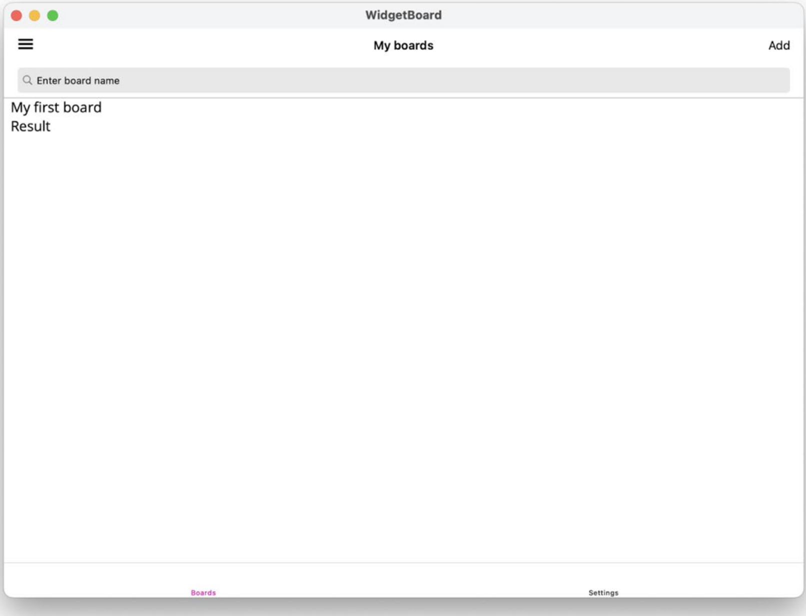

Figure 6-9. *The application showing the list of boards*

This concludes our chapter on Shell. I really hope each of the carefully crafted examples shows how you can achieve a variety of different scenarios.

Summary

It is worth stating that anything you do with Shell is built out of components in the .NET MAUI box. Shell puts them together in an opinionated way, but you can use all of those things separately, outside of Shell as well if that's what you want.

In this chapter, you have

- Added some placeholder pages and view models
- Gained an understanding of Shell and applied this to building your application's structure
- Applied the Shell navigation to allow you to navigate to your next page and the next chapter
- Built your flyout menu using all the learnings in this chapter
- Added tabs into the application
- Added the ability to search for boards
- Introduced the ability to add a new board by showing a page and waiting for a result

In the next chapter, you will

- Create your own layout
- Make use of a variety of options when adding bindable properties
- Provide command support from your layout
- Use your layout in your application

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch06`.

Extra Assignment

This extra assignment is a culmination of the last two chapters combined. I would like you to consider how you might add a second layout type (e.g., a board where widgets could be placed anywhere) given that you

- Have a single layout type on your `BoardDetailsPage`
- Have options displayed when this type is selected
- Pass a `FixedLayout` instance over as data to your `FixedBoardPage`

I would love to see what concepts you come up with.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch06-extra.

CHAPTER 7

Creating Our Own Layout

Abstract

In Chapter 5, you learned a lot of the fundamentals of building and binding your user interfaces. In this chapter, you will create your own layout, make use of a variety of options when adding bindable properties, provide command support from your layout, and make use of your layout in your application. This will serve as the basis for adding much more functionality as we cover a variety of different topics in future chapters.

Let's recap what you achieved in the last chapter: you provided the ability for a user to create a board and supply a number of columns and rows. You now need to lay out your board with the number of columns and rows the user has configured and populate widgets onto the board. Figure 7-1 is a mock-up of what you will achieve by the end of this chapter.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_7

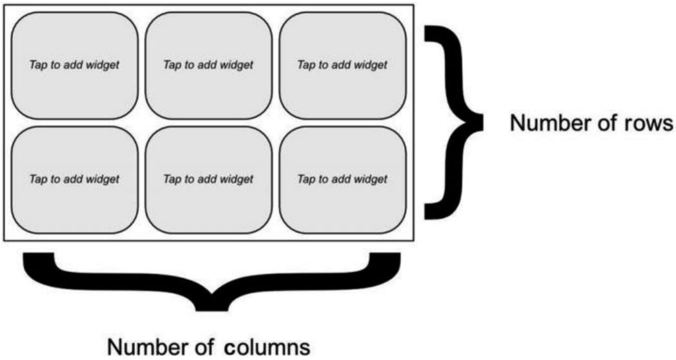

Figure 7-1. *Mockup of a board*

At the end of the last chapter, I discussed the idea of having a second type of layout in the “Extra Assignment” section. To continue with this theme, I have structured the architecture of the layout to aid in this journey. I am a fan of taking an approach like this because it allows you to potentially replace one part of the implementation without impacting the others.

`BoardLayout` will be responsible for displaying the widgets. It will be assigned an `ILayoutManager` implementation, which will decide where to place the widgets. You will be adding a `FixedLayoutManager` to decide this part.

Placeholder

The first item that you need to create is the placeholder to show where a widget will be placed. There isn’t too much to this control, but creating it allows you to group all of the related bits and pieces together. Figure 7-2 shows what your `Placeholder` control will look like when rendered inside the application.

Figure 7-2. *Mock-up of the Placeholder control*

In order to achieve the above look, you are going to make use of the Border control. This is a really useful control. It allows you to provide borders, custom corner radius, shadows, and other styling options. It also behaves much like the ContentView in that it can contain a single child control.

Create a folder called Controls in your main project. It will house the Placeholder control and potentially more as you build your application.

Next, add a new class to the folder and call it Placeholder. Note that you are opting to create the control purely in C# without XAML; the main reason is that it results in less code. I always find there is never a single way to build things, and even if you like XAML, at times it doesn't add any value, just like in this scenario. Of course, if you prefer to build your UI with XAML, you can do so.

```
namespace WidgetBoard.Controls;

public class Placeholder : Border
{
    public Placeholder()
    {
        Content = new Label
```

```
        {
            Text = "Tap to add widget",
            FontAttributes = FontAttributes.Italic,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
    }

    public int Position { get; set; }
}
```

As discussed, there isn't too much to this implementation, but let's still break it down. Here you have

- Created a control that inherits from `Border`
- Set the content of your control to be a `Label` showing fixed text in an italic font and the text is centered both horizontally and vertically
- Added a `Position` property to know where in the layout it will be positioned

Now you can start building the layout that will display the placeholders and ultimately your widgets.

ILayoutManager

You have a slight chicken-and-egg scenario here. You need to create a board and a layout manager, both of which need to know about the other; therefore, let's add in the LayoutManager parts first.

The purpose of the `ILayoutManager` interface is to define how the `BoardLayout` will interact with a layout manager implementation.

Create a folder called `Layouts` in your main project. It will house the `ILayoutManager` interface and more as you build your application.

Next, add a new class to the folder and call it ILayoutManager.

```
namespace WidgetBoard.Layouts;

public interface ILayoutManager
{
    object BindingContext { get; set; }
    BoardLayout? Board { get; set; }
    void SetPosition(BindableObject bindableObject,
    int position);
}
```

Let's break it down so you have a clear definition of what you just created:

- The BindingContext property allows you to pass the context down from the BoardLayout later. This is important for allowing bindings on the layout manager.
- The Board property allows the manager to interact directly with the board it is intended to assist.
- The SetPosition method allows the manager to use the position parameter and set the appropriate layout settings on the widget/placeholder.

BoardLayout

Your BoardLayout will be the parent of your widgets. Create the layout inside your Layouts folder.

- Right-click the Layouts folder.
- Select Add ➤ New Item.
- Select the **.NET MAUI** tab.

- Select the **.NET MAUI ContentView (XAML)** option.
- Enter the name *BoardLayout.*
- Click **Add**.

This will give you two files. You'll modify each one individually.

BoardLayout.xaml

Modify the existing contents to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<Grid
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WidgetBoard.Layouts.BoardLayout"
    x:Name="Self">

    <Grid
        x:Name="PlaceholderGrid" />

    <Grid
        x:Name="WidgetGrid"
        ChildAdded="OnWidgetsChildAdded"
        BindableLayout.ItemsSource="{Binding ItemsSource,
        Source={x:Reference Self}}"
        BindableLayout.ItemTemplateSelector="{Binding
        ItemTemplateSelector, Source={x:Reference Self}}"
        InputTransparent="True"
        CascadeInputTransparent="False" />
</Grid>
```

You have added quite a bit to this that might not feel familiar, so again let's break it down.

Your main layout is a `Grid`, and inside of it are two more `Grids`.

The first inner `Grid` (`PlaceholderGrid`) is where you add the `Placeholder` control you created earlier in this chapter.

The second inner `Grid` (`WidgetGrid`) is where you add widgets. The reason you have built the control this way is mainly so you can utilize a really impressive piece of functionality that drastically reduces the amount of code you have to write: `BindableLayout`.

You have not supplied a `Grid.Row` or `Grid.Column` to either of your inner `Grids`. This results in both controls filling the space of the parent `Grid` and the second one overlapping the first. This behavior can provide some real power when building rather complex UIs.

BindableLayout

`BindableLayout` allows you to turn a layout control into a control that can be populated by a collection of data. `BindableLayout` is not a control itself, but it provides the ability to enhance layout controls by adding an `ItemsSource` property for bindings. This means that all of the layouts you learned about in the previous chapter (e.g., `Grid`, `AbsoluteLayout`, `FlexLayout`, `HorizontalStackLayout`, `VerticalStackLayout`) can be turned into a layout that can show a specific set of controls for each item that is provided. For this, you need to set two properties:

- `BindableLayout.ItemsSource`: This is the collection of items that you wish to represent in the UI.
- `BindableLayout.ItemTemplate` or `BindableLayout.ItemTemplateSelector`: This allows you to define how the item will be represented. In most scenarios, `ItemTemplate` is enough, but this only works when you have one type of item to display in your collection. If you have multiple types, each widget will be a separate type in your application, so you need to use the `ItemTemplateSelector`.

I won't actually be providing the source for these bindings just yet; this will be done in Chapter 8. For now, you just need to make it possible to bind them.

BoardLayout.xaml.cs

Now that you have created your XAML representation, you need to add in the code-behind, which will work with it. We are going to follow a slightly different approach for this and the next section; you have a lot of code to add now so you will add it in stages and we will talk around what you are adding.

The initial code should look as follows:

```
namespace WidgetBoard.Layouts;

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

Adding the LayoutManager Property

You want to allow the consumer of your `BoardLayout` control to be able to supply a `LayoutManager` that will control where the widgets are placed. For this, you need to add the following:

```
private ILayoutManager? layoutManager;

public ILayoutManager? LayoutManager
{
    get => layoutManager;
    set
```

```
    {
        layoutManager = value;

        if (layoutManager is not null)
        {
            layoutManager.Board = this;
        }
    }
}
```

The key detail of this implementation is how it assigns the `Board` property on the `LayoutManager` to your `BoardLayout` control. This is to allow the manager to interact with the layout.

One very important thing to consider is that when you create properties that can be set in XAML, their setters can be called before your control has its `BindingContext` property set. Therefore, you usually need to handle both scenarios when relying on both pieces of functionality. To give a concrete example of this, you have your `LayoutManager` property that you have added. It will allow you to set bindings on it also, but it won't have a `BindingContext` passed down. For this, you need to override the `OnBindingContextChanged` method in your `BoardLayout` class and assign the value to your `LayoutManager`.

```
protected override void OnBindingContextChanged()
{
    base.OnBindingContextChanged();

    if (layoutManager is not null)
    {
        layoutManager.BindingContext = this.BindingContext;
    }
}
```

Adding the ItemsSource Property

Your `BoardLayout` also needs to accept a collection of widgets that will ultimately be displayed on screen. For controls that support displaying multiple items, the common name used for such a property is `ItemsSource`. So add a property with that name. You will need to add the following to the top of the file:

```
using System.Collections;
```

This is to allow you to use the `IEnumerable` type.

```
public static readonly BindableProperty ItemsSourceProperty =
    BindableProperty.Create(
        nameof(ItemsSource),
        typeof(IEnumerable),
        typeof(BoardLayout));

public IEnumerable ItemsSource
{
    get => (IEnumerable)GetValue(ItemsSourceProperty);
    set => SetValue(ItemsSourceProperty, value);
}
```

In the majority of scenarios, you bind an `ObservableCollection` to an `ItemsSource` property, which is of a different type to `IEnumerable`. By choosing to use `IEnumerable`, it allows the consumers of your layout to provide any type that supports holding multiple items. This means that you can supply an `ObservableCollection` or you can supply a `List`.

Adding the ItemTemplateSelector Property

Now that you have a collection of items to display on screen, you need to know how to display them. It can be common to see controls that have an `ItemsSource` property also have an `ItemTemplate` or an

ItemTemplateSelector or even both properties. An ItemTemplate allows a developer to define how each item in the ItemsSource will be rendered on screen. The reason you aren't using this approach is because you can only define one template for all items. You will be binding your widget view models to the ItemsSource property, which means you will have several different views that you will want to display. This is where the ItemTemplateSelector property comes in.

```
public static readonly BindableProperty
ItemTemplateSelectorProperty =
    BindableProperty.Create(
        nameof(ItemTemplateSelector),
        typeof(DataTemplateSelector),
        typeof(BoardLayout));

public DataTemplateSelector ItemTemplateSelector
{
    get => (DataTemplateSelector)GetValue(ItemTemplateSelector
    Property);
    set => SetValue(ItemTemplateSelectorProperty, value);
}
```

You make use of the DataTemplateSelector type for your property here. You will create an implementation a little later in this chapter, but for now, it allows you to override the OnSelectTemplate method and provide a suitable template for the item that is passed in.

Handling the ChildAdded Event

I discussed earlier how the BindableLayout feature allows you to populate a control with multiple views based on bindings. You need to hook into the ChildAdded event so that your LayoutManager implementation can determine where the new child should be positioned.

```
private void OnWidgetsChildAdded(object sender,
ElementEventArgs e)
{
    if (e.Element is IWidgetView widgetView)
    {
        LayoutManager.SetPosition(e.Element, widgetView.
        Position);
    }
}
```

This handler checks to see if the new child being added is of the `IWidgetView` type, and if it is, it delegates out to the `LayoutManager` implementation to set the widget's position.

Adding Remaining Bits

You have a few extra methods and properties to add in that will be used by the `FixedLayoutManager`. Let's add them and discuss their purpose as you go.

Add the `using` statement at the top of the file:

```
using WidgetBoard.Controls;
```

Then add the first new method:

```
public void AddPlaceholder(Placeholder placeholder) =>
PlaceholderGrid.Children.Add(placeholder);
```

This method allows the caller to pass a placeholder that will be added to `PlaceholderGrid`. This is useful when first loading a board or when dealing with a widget being removed from a specific position.

```
public void RemovePlaceholder(Placeholder placeholder) =>
PlaceholderGrid.Children.Remove(placeholder);
```

This method allows the caller to pass a placeholder that will be removed from the PlaceholderGrid. This is useful for when dealing with a widget being added to a specific position.

```
public void AddColumn(ColumnDefinition columnDefinition)
{
    PlaceholderGrid.ColumnDefinitions.Add(columnDefinition);
    WidgetGrid.ColumnDefinitions.Add(columnDefinition);
}
```

This method allows for the board's columns to be defined on both the PlaceholderGrid and WidgetGrid.

```
public void AddRow(RowDefinition rowDefinition)
{
    PlaceholderGrid.RowDefinitions.Add(rowDefinition);
    WidgetGrid.RowDefinitions.Add(rowDefinition);
}
```

This method allows for the board's rows to be defined on both the PlaceholderGrid and WidgetGrid.

```
public IReadOnlyList<Placeholder> Placeholders =>
PlaceholderGrid.Children.OfType<Placeholder>().ToList();
```

This property provides all children from the PlaceholderGrid that are of type Placeholder. This is to allow for determining which placeholder needs to be removed when adding a widget.

FixedLayoutManager

The final part for you to create is the FixedLayoutManager class. This will provide the logic to

- Accept the number of rows and columns for a board
- Provide tap/click support through a command

- Build the board layout
- Set the correct row/column position for each widget

Create the file - add a new class to the *Layouts* folder and call it FixedLayoutManager and then you can work through adding each of the above pieces of functionality. Let's add a new class file and call it FixedLayoutManager.cs. Add the following content:

```
namespace WidgetBoard.Layouts;

public class FixedLayoutManager
{
}
```

To start, you are going to want to add the following using statements:

```
using System.Windows.Input;
using WidgetBoard.Controls;
```

And also make your class inherit from BindableObject and implement your ILayoutManager interface. Your class should now look as follows:

```
using System.Windows.Input;
using WidgetBoard.Controls;

namespace WidgetBoard.Layouts;

public class FixedLayoutManager : BindableObject,
ILayoutManager
{
}
```

The reason for inheriting from BindableObject is down to the fact that you need to add some bindable properties onto this class so that developers using this implementation can bind values to the properties.

Accepting the Number of Rows and Columns for a Board

You need to add the ability to set the number of rows and columns to be displayed in your fixed layout board. For this, you are going to add two bindable properties to your `FixedLayoutManager` class.

Adding the NumberOfColumns Property

```
public static readonly BindableProperty
NumberOfColumnsProperty =
    BindableProperty.Create(
        nameof(NumberOfColumns),
        typeof(int),
        typeof(FixedLayoutManager),
        defaultBindingMode: BindingMode.OneWay,
        propertyChanged: OnNumberOfColumnsChanged);

public int NumberOfColumns
{
    get => (int)GetValue(NumberOfColumnsProperty);
    set => SetValue(NumberOfColumnsProperty, value);
}

private static void OnNumberOfColumnsChanged(BindableObject
bindable, object oldValue, object newValue)
{
    var manager = (FixedLayoutManager)bindable;
    manager.InitializeGrid();
}
```

The key difference with this implementation over the previous bindable properties that you created is the use of the `propertyChanged` parameter. It allows you to define a method (see `OnNumberOfColumnsChanged`) that will be called whenever the property value changes.

The property changed method will only be called when the value changes. This means that it may not be called initially if the value does not change from the default value. You will also notice how we are casting the `bindable` parameter to `FixedLayoutManager`; this is because we have to declare the method as static and therefore do not have access to the instance the `BindableProperty` belongs to. The owner of the property is passed in via the `bindable` parameter to our method, hence the need to make our `FixedLayoutManager` class inherit from `BindableObject`.

Adding the NumberOfRows Property

```
public static readonly BindableProperty NumberOfRowsProperty =
    BindableProperty.Create(
        nameof(NumberOfRows),
        typeof(int),
        typeof(FixedLayoutManager),
        defaultBindingMode: BindingMode.OneWay,
        propertyChanged: OnNumberOfRowsChanged);

public int NumberOfRows
{
    get => (int)GetValue(NumberOfRowsProperty);
    set => SetValue(NumberOfRowsProperty, value);
}
```

```
private static void OnNumberOfRowsChanged(BindableObject
bindable, object oldValue, object newValue)
{
    var manager = (FixedLayoutManager)bindable;
    manager.InitializeGrid();
}
```

This is virtually identical to the `NumberOfColumns` property that you just added, except for the `NumberOfRows` value.

Providing Tap/Click Support Through a Command

The next item on your list is to provide the ability to handle tap/click support. This is your first time providing command support; you used commands in your bindings, but that was on the source side rather than the target side like here.

First, you need to add the bindable property, which should start to feel rather familiar.

```
public static readonly BindableProperty
PlaceholderTappedCommandProperty =
    BindableProperty.Create(
        nameof(PlaceholderTappedCommand),
        typeof(ICommand),
        typeof(FixedLayoutManager));

public ICommand PlaceholderTappedCommand
{
    get => (ICommand)GetValue(PlaceholderTappedCommand
    Property);
    set => SetValue(PlaceholderTappedCommandProperty, value);
}
```

Next, you need to add the code that will execute the command. You will be relying on the use of a TapGestureRecognizer by adding one to your Placeholder control inside your InitializeGrid method that you will be adding in the next section. For now, you can add the method that will be used so that you can focus on how to execute the command. Let's add the code and then look over the details.

```
private void OnTapGestureRecognizerTapped(object? sender,
EventArgs e)
{
    if (sender is Placeholder placeholder)
    {
        if (PlaceholderTappedCommand?.CanExecute(placeholder.
        Position) == true)
        {
            PlaceholderTappedCommand.Execute(placeholder.
            Position);
        }
    }
}
```

You can see from the implementation that there are three main parts to the command execution logic:

- First, you make sure that command has a value.
- Second, you check that you can execute the command. If you recall back in Chapter 5, you provided a method to prevent the command from executing if the user hadn't entered a BoardName.
- Finally, you execute the command and pass in the command parameter. For this scenario, you will be passing in the current position of the placeholder so when a widget is added, it can be placed in the same position.

Building the Board Layout

Now you can focus on laying out the underlying Grids so that they display as per the user's entered values for rows and columns.

First, add in a property to store the current Board because you need to use it when building the layout. You also need to record whether you have built the layout to prevent any unnecessary updates rebuilding the user interface.

```
private BoardLayout? board;
private bool isInitialized;

public BoardLayout? Board
{
    get => board;
    set
    {
        board = value;
        InitializeGrid();
    }
}
```

Your method to build the grid layout has several parts, so let's add them as you go and discuss their value. You initially need to make sure that you have valid values for the Board, NumberOfRows, and NumberOfColumns properties plus you haven't already built the UI.

```
private void InitializeGrid()
{
    if (Board is null ||
        NumberOfColumns == 0 ||
        NumberOfRows == 0 ||
        isInitialized == true)
```

```
    {
        return;
    }
    isInitialized = true;
}
```

The next step is to use the NumberOfColumns value and add it to your Board. Let's add this to the end of the InitializeGrid method.

```
for (int i = 0; i < NumberOfColumns; i++)
{
    Board.AddColumn(new ColumnDefinition(new GridLength
    (1, GridUnitType.Star)));
}
```

The GridUnitType.Star value means that each column will have an even share of the width of the grid. So if the Grid is 300 pixels wide and you have three columns, then each column has a resulting width of 100 pixels.

The next step is to use the NumberOfRows value and add it to your Board. Let's add this to the end of the InitializeGrid method.

```
for (int i = 0; i < NumberOfRows; i++)
{
    Board.AddRow(new RowDefinition(new GridLength
    (1, GridUnitType.Star)));
}
```

The final step in your InitializeGrid method is to populate each cell (row and column) combination with a Placeholder control.

```
for (int column = 0; column < NumberOfColumns; column++)
{
    for (int row = 0; row < NumberOfRows; row++)
    {
        var placeholder = new Placeholder();
```

```
            placeholder.Position = row * NumberOfColumns + column;
            var tapGestureRecognizer = new TapGestureRecognizer();
            tapGestureRecognizer.Tapped += OnTapGestureRecognizer
            Tapped;
            placeholder.GestureRecognizers.Add(tapGesture
            Recognizer);
            Board.AddPlaceholder(placeholder);
            Grid.SetColumn(placeholder, column);
            Grid.SetRow(placeholder, row);
        }
}
```

In the above code, you

- Looped through the combinations of rows/columns
- Created a Placeholder control
- Set its position for use later
- Added a TapGestureRecognizer to handle user interaction
- Added the Placeholder to the Board
- Positioned the Placeholder to the correct column and row position

Setting the Correct Row/Column Position for Each Widget

The final part in building the board layout is to provide the method required by the ILayoutManager interface that your FixedLayoutManager is implementing. This method will

- Calculate the column/row value based on the `position` parameter passed in
- Position the `bindableObject` parameter passed into the calculated column and row position
- Remove any existing `Placeholder` in the position

```
public void SetPosition(BindableObject bindableObject, int
position)
{
    if (NumberOfColumns == 0 || Board is null)
    {
        return;
    }

    int column = position % NumberOfColumns;
    int row = position / NumberOfColumns;

    Grid.SetColumn(bindableObject, column);
    Grid.SetRow(bindableObject, row);

    var placeholder = Board.Placeholders.FirstOrDefault
    (p => p.Position == position);
    if (placeholder is not null)
    {
        Board.RemovePlaceholder(placeholder);
    }
}
```

Now that you have completed the work of providing a `BoardLayout` and managing its layout with your `FixedLayoutManager` class, you should go ahead and use it in your application.

Using Your Layout

Before you can jump in and start using the BoardLayout you have created, there is a little bit more work to be done. You need to

- Add a factory that will create instances of your widgets
- Add in the DataTemplateSelector that I referred to earlier on
- Update your FixedBoardPageViewModel so your bindings will work

Adding a Factory That Will Create Instances of Your Widgets

For this, you are going to create a new class called WidgetFactory in the root of your project.

```
using WidgetBoard.ViewModels;
using WidgetBoard.Views;

namespace WidgetBoard;

public class WidgetFactory
{
}
```

There are three main purposes for this factory:

- Allows for the registration of widget views and view models
- Creation of a widget view
- Creation of a widget view model

So let's support these three requirements.

Allowing for the Registration of Widget Views and View Models

You need to add the following code:

```
private static IDictionary<Type, Type> widgetRegistrations =
new Dictionary<Type, Type>();

private static IDictionary<string, Type>
widgetNameRegistrations = new Dictionary<string, Type>();

public static void RegisterWidget<TWidgetView,
TWidgetViewModel>(string displayName) where TWidgetView :
IWidgetView where TWidgetViewModel : IWidgetViewModel
{
    widgetRegistrations.Add(typeof(TWidgetViewModel),
    typeof(TWidgetView));
    widgetNameRegistrations.Add(displayName,
    typeof(TWidgetViewModel));
}

public IList<string> AvailableWidgets =>
widgetNameRegistrations.Keys.ToList();
```

The above may look a little complicated, but if you break it down, hopefully it should become clear. You have added two fields that will store the type and name information needed for when you create the instances of widgets.

The `RegisterWidget` method takes a display name parameter and two types:

- `TWidgetView`: This must implement your `IWidgetView` interface.
- `TWidgetViewModel`: This must implement your `IWidgetViewModel` interface.

You then store a mapping between the view model type and the view type (`widgetRegistrations`). This allows you to create a view when you pass in a view model. This really helps you to keep a clean separation between your view and view model.

You also store a mapping between the display name and the view model type (`widgetNameRegistrations`). This will allow you to present an option on screen to the user. Once they choose the name of the widget they would like to add, the factory will create an instance of it.

Creation of a Widget View

You first need to add a dependency to your constructor.

```
private readonly IServiceProvider serviceProvider;

public WidgetFactory(IServiceProvider serviceProvider)
{
    this.serviceProvider = serviceProvider;
}
```

The `IServiceProvider` will allow you to create a new instance of your widgets and make sure that they are provided with all of their dependencies. Don't worry about needing to register the `IServiceProvider` implementation with your `MauiAppBuilder` as you have done with other dependencies that you require. This is automatically provided by .NET MAUI.

Now let's add the ability to create the widget view.

```
public IWidgetView? CreateWidget(IWidgetViewModel
widgetViewModel)
{
    if (widgetRegistrations.TryGetValue(widgetViewModel.
    GetType(), out var widgetViewType))
    {
```

```
        var widgetView = (IWidgetView)serviceProvider.
        GetRequiredService(widgetViewType);
        widgetView.WidgetViewModel = widgetViewModel;
        return widgetView;
    }
    return null;
}
```

Breaking this down:

- You check whether the supplied `widgetViewModels` type has been registered with the factory.
- If it has, you use the `IServiceProvider` to get an instance of the associated widget view.
- You assign the `widgetViewModel` parameter value to the `WidgetViewModel` property on the widget view. This is to allow for the setting of the widget's `BindingContext` property.

Creation of a Widget View Model

You also need to provide the ability to create the widget view model because this is required in your view model.

```
public IWidgetViewModel? CreateWidgetViewModel(string
displayName)
{
    if (widgetNameRegistrations.TryGetValue(displayName, out
    var widgetViewModelType))
    {
        return (IWidgetViewModel)serviceProvider.GetRequired
        Service(widgetViewModelType);
```

```
    }
    return null;
}
```

Breaking this down:

- You check whether the supplied `displayName` has been registered with the factory.
- If it has, you use the `IServiceProvider` to get an instance of the associated widget view model.

Registering the Factory with MauiAppBuilder

Inside your `MauiProgram.cs` file, you need to register your `WidgetFactory` with the `MauiAppBuilder` to make sure any dependencies can resolve it. Open that file and add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton<WidgetFactory>();
```

Registering Your ClockWidget with the Factory

Now that you have your WidgetFactory, you need to modify it so that the factory can create the widget for you. This requires a number of steps, so let's walk through it.

First, open the `ClockWidgetView.xaml.cs` file and change it to the following:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public partial class ClockWidgetView : Label, IWidgetView
{
    public ClockWidgetView(ClockWidgetViewModel
    clockWidgetViewModel)
```

```
    {
        InitializeComponent();
        WidgetViewModel = clockWidgetViewModel;
        BindingContext = clockWidgetViewModel;
    }

    public IWidgetViewModel WidgetViewModel { get; set; }
}
```

This results in your `ClockWidgetView` taking a dependency on `ClockWidgetViewModel`.

Next, you need to register your widget with the factory. Open your `MauiProgram.cs` file and add the following lines to the `CreateMauiApp` method:

```
WidgetFactory.RegisterWidget<ClockWidgetView, ClockWidgetView
Model>("Clock");
builder.Services.AddTransient<ClockWidgetView>();
builder.Services.AddTransient<ClockWidgetViewModel>();
```

This will enable the `WidgetFactory` to return the clock widget as an option when presented in your overlay.

WidgetTemplateSelector

The main purpose of this implementation is to provide a conversion between the widget view models that you will be storing on your `FixedBoardPageViewModel` and something that can actually be rendered on the screen. You are going to depend on the `WidgetFactory` you have just created. Create the class under the root project folder and modify its contents to the following:

```
using WidgetBoard.ViewModels;
```

```
namespace WidgetBoard;

public class WidgetTemplateSelector : DataTemplateSelector
{
    private readonly WidgetFactory widgetFactory;

    public WidgetTemplateSelector(WidgetFactory widgetFactory)
    {
        this.widgetFactory = widgetFactory;
    }

    protected override DataTemplate? OnSelectTemplate(object
    item, BindableObject container)
    {
        if (item is IWidgetViewModel widgetViewModel)
        {
            return new DataTemplate(() => widgetFactory.
            CreateWidget(widgetViewModel));
        }
        return null;
    }
}
```

The main part you need to focus on here is the `OnSelectTemplate` method. I did discuss the purpose of this method briefly earlier on; let's take a deeper look now. Its main purpose is to provide a `DataTemplate`, which, as its name suggests, is a template for a piece of data. Using a `DataTemplate` will result in something that can be rendered on screen each time that piece of data is added. This is a great way to keep the separation between view and view model – the view model holds the data and the view knows about the template which represents how to display that data.

In your implementation, you can see that

- You check whether the item passed in implements your `IWidgetViewModel` interface.
- If so, then you create a new `DataTemplate` and rely on the `WidgetFactory` to return the widget view that is mapped to the view models type.

Registering the Template Selector with MauiAppBuilder

Inside your `MauiProgram.cs` file, you need to register your `WidgetTemplateSelector` with the `MauiAppBuilder` to make sure any dependencies can resolve it. Open that file and add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton<WidgetTemplateSelector>();
```

Updating FixedBoardPageViewModel

You need to add in the properties that you can bind to in your view.

```
private string boardName = string.Empty;
private int numberOfColumns;
private int numberOfRows;

public string BoardName
{
    get => boardName;
    set => SetProperty(ref boardName, value);
}
```

```
public int NumberOfColumns
{
    get => numberOfColumns;
    set => SetProperty(ref numberOfColumns, value);
}

public int NumberOfRows
{
    get => numberOfRows;
    set => SetProperty(ref numberOfRows, value);
}

public ObservableCollection<IWidgetViewModel> Widgets { get; }

public WidgetTemplateSelector WidgetTemplateSelector { get; }
```

Notice that the `Widgets` and `WidgetTemplateSelector` properties do not call the `SetProperty` method to notify the UI of changes. This is a perfectly valid scenario. You know that the value will be set in the constructor, and therefore, the value will be set before the binding is applied.

You also need to add in the remaining code to your `ApplyQueryAttributes` method that you added in the last chapter. It should now look like the following:

```
public void ApplyQueryAttributes(IDictionary<string,
object> query)
{
    var board = (Board)query["Board"];

    BoardName = board.Name;
    NumberOfColumns = board.NumberOfColumns;
    NumberOfRows = board.NumberOfRows;
}
```

Finally, you need to add the `WidgetTemplateSelector` as a dependency in your constructor. It should now look like the following:

```
public FixedBoardPageViewModel(
    WidgetTemplateSelector widgetTemplateSelector
)
{
    WidgetTemplateSelector = widgetTemplateSelector;
    Widgets = [];
}
```

You are now ready to add the layout to your page.

Finally Using the Layout

Now that you have built your layout, you should go ahead and use it. You previously added the `FixedBoardPage` so you can go ahead and change it to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:layouts="clr-namespace:WidgetBoard.Layouts"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Pages.FixedBoardPage"
    Title="FixedBoardPage"
    x:DataType="viewModels:FixedBoardPageViewModel">
    <layouts:BoardLayout
        ItemsSource="{Binding Widgets}"
        ItemTemplateSelector="{Binding WidgetTemplate
        Selector}">
        <layouts:BoardLayout.LayoutManager>
```

```
            <layouts:FixedLayoutManager
                NumberOfColumns="{Binding NumberOfColumns}"
                NumberOfRows="{Binding NumberOfRows}" />
        </layouts:BoardLayout.LayoutManager>
    </layouts:BoardLayout>
</ContentPage>
```

This now includes your shiny new `BoardLayout` complete with all the bindings you have created to make it functional.

One additional change you will need to make is to link the `FixedBoardPage` to the `FixedBoardPageViewModel`; to do this, you can open the *FixedBoardPage.xaml.cs* file and modify the contents to the following; the changes are in **bold**:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Pages;

public class FixedBoardPage : ContentPage
{
    public FixedBoardPage(FixedBoardPageViewModel viewModel)
    {
        InitializeComponent();
        BindingContext = viewModel;
    }
}
```

This sets us up nicely for when we start to use the page and display it in our application.

Summary

In this chapter, you have

- Created your own layout
- Made use of a variety of options when adding bindable properties
- Provided command support from your layout
- Used your layout in your application

In the next chapter, you will

- Gain an understanding of what accessibility is
- Learn why it is important to build inclusive applications
- Look at how you can make use of .NET MAUI functionality
- Consider other scenarios and how to support them
- Look over some testing options to support your journey to building accessible applications

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch07`.

Extra Assignment

You will have noticed how a lot of the naming includes the word Fixed. Let's continue the extra assignment from the previous chapter and build a board that is a variation of this approach. I really like the idea of a freeform board where the user can position their widgets wherever they like. This is a little more involved, but if you consider how the `BoardLayout` can use `AbsoluteLayouts` rather than `Grids`, then a new `ILayoutManager` implementation should hopefully be where the alternative logic will need to be applied. If you do embark on this journey, please feel free to share your experience and findings.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch07-extra`.

CHAPTER 8

Accessibility

Abstract

In this chapter, you will be taking a break from adding new parts to the user interface in order to gain an understanding of what accessibility is, why you should make your applications accessible, and how .NET MAUI makes this easier. You will also cover some testing options to support your journey to building accessible applications.

I wanted this chapter to appear earlier on in this book. I feel it is such an important topic and one that you really do need to consider early on in your projects. It has come to settle nicely in the middle of the book now because you needed some UI to apply the concepts to.

What Is Accessibility?

The definition of accessibility according to the Cambridge Dictionary (`https://dictionary.cambridge.org/dictionary/english/accessibility`) is

> *"the quality of being easy to understand"*

By considering the scenarios where your application might be less easy to understand for a large percentage of the world's population that have some form of disability, you can learn to provide ways to break down the complexities in understanding the content. This might be through the use

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_8

of assistive technologies such as voice-over assistants or screen readers, or even providing the ability to increase the font size to make the content easier to read.

All of this can help you as a developer learn how to build applications that are much more inclusive of the entire population of the world.

Why Make Your Applications Accessible?

I heard an excellent quote recently, and sadly I have been unable to discover the original author of the quote, but it is *"if you don't know whether your application is accessible, then you can safely say that it is not."* Essentially, if you are not putting any effort into making it accessible, then you can almost guarantee that it is not.

According to the World Health Organization, *globally at least 2.2 billion people have a near or distance vision impairment* (`www.who.int/news-room/fact-sheets/detail/blindness-and-visual-impairment`).

You want to build your applications and make them as successful as possible. Imagine immediately ruling out up to 27% of your potential market purely based on not making your application more inclusive for that population.

What to Consider When Making Your Applications Accessible

There is a whole heap of things you can do in order to make your applications more inclusive. To aid you on your journey to building accessible applications, there is a fantastic set of guidelines known as the *Web Content Accessibility Guidelines (WCAG)*. There are four main principles to consider:

- Perceivable: Making sure that you provide information that can be perceived by the user. This can be by providing text-based alternatives to images, suitable contrast ratios, adaptive text sizing, and much more.
- Operable: Making sure that you provide the user with the ability to use the application. This can be by providing keyboard navigation, making sure they have enough time to read and use the content, and much more.
- Understandable: Making sure that you provide a user interface that is understandable to the user. This can be making sure that the content is readable, predictable (appear and behave as expected), and helps the user avoid making mistakes.
- Robust: Making sure the content is robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies. This can be by providing suitable support for assistive technologies.

To read more on these guidelines, I thoroughly recommend checking out the Quick Reference Guide at `www.w3.org/WAI/WCAG21/quickref/`.

How to Make Your Application Accessible

There are several things to consider when building an application that is inclusive. This section will not provide a complete set of tools for building applications inclusive for all. However, it will provide some insights to what .NET MAUI offers and some other concepts to consider to set you off on a journey of discovery to building much more accessible applications.

Screen Reader Support

.NET MAUI provides great tools to provide explicit support for the screen readers on each of the supported platforms. I feel it is worth highlighting that point again: **.NET MAUI utilizes the screen readers on each platform**. This means that they will need to be enabled by the user for the settings to take effect. You will dive into each concept and how it enables you to expose information to those screen readers so you can provide a much more informative experience for your users.

As a starting exercise, pick up your phone and turn on your screen reader assistant.

Enabling the Screen Reader on iOS

In order to enable the screen reader or VoiceOver feature on iOS, you can follow these steps:

1. Navigate to the Settings app.
2. Select **Accessibility**.
3. Select **VoiceOver**.
4. Enable the switch to turn the feature on.

Enabling the Screen Reader on Android

In order to enable the screen reader or TalkBack feature on Android, you can follow these steps:

1. Navigate to the Settings app.
2. Select **Accessibility**.
3. Select **Screen reader** (this might appear as **TalkBack** on your device).

4. Enable the **Voice Assistant** (this might appear as **TalkBack** on your device).

Try Using the Screen Reader

Try navigating around to get an understanding of what the experience is like and, most importantly, try an application you built. Does it provide a good experience? I highly doubt it.

Let's see how you can make the WidgetBoard application more accessible with the screen readers available. Thankfully you haven't built too much UI already, so you are in a good position to start. I urge you to consider applying concepts like this as early on in the development phase as possible.

SemanticProperties

The `SemanticProperties` class offers a set of attached properties that can be applied to any visual element. .NET MAUI applies these property values on the platform-specific APIs that provide accessibility.

Let's look through each of the properties and apply them to your `BoardDetailsPage`.

SemanticProperties.Description

The `SemanticProperties.Description` property allows you to define a short string that will be used by the screen reader to announce the element to the user when it gains focus. This should be a name that implies the intent of the element if the user were to interact with it.

As I type this chapter, I am testing the application. The first `Entry` added on the `BoardDetailsPage` currently results in the macOS VoiceOver assistant announcing *"edit text, is editing, blank"*.

You can change the `Entry` to the following:

```
<Entry
    Text="{Binding BoardName}"
    SemanticProperties.Description="Enter the board name"/>
```

This now results in *"Enter the board name, is editing, blank"* being announced, which is much more useful to the user.

You can take this a step further. You have a label above that just has the Text of "Name." If you change this to use your new descriptive text, then you can set the `SemanticProperties.Description` value to its text. Let's do that now; the changes are highlighted in **bold**:

```
<Label
    Text="Enter the board name"
    x:Name="EnterBoardNameLabel"
    FontAttributes="Bold" />
<Entry
    Text="{Binding BoardName}"
    SemanticProperties.Description="{Binding Text,
    Source={x:Reference EnterBoardNameLabel}}" />
```

The resulting code may look less appealing, but it provides a number of benefits:

- The text description is more informative on the Label.
- When you add in localization support, you will have only one text field to update.

The macOS screen reader does provide a second announcement following the announcement you have been improving. This follow-up is *"You are currently on a text field. To enter text in this field, type."* This isn't the most informative, so let's provide a better hint to the user.

The act of setting the `SemanticProperties.Description` property will automatically make a visual element be announced by the screen reader. By default, an `Image` control is not announced, but by setting this property, the text will be announced when the control gains semantic focus.

SemanticProperties.Hint

The `SemanticProperties.Hint` property allows you to provide a string that the screen reader will announce to the user so that they have a better understanding of the purpose of the control.

Let's add a hint to `Entry` with the addition in **bold**:

```
<Entry
    Text="{Binding BoardName}"
    SemanticProperties.Description="{Binding Text,
    Source={x:Reference EnterBoardNameLabel}}"
    SemanticProperties.Hint="Provides a name that will be
    used to identify your widget board. This is a required
    field." />
```

This change results in *"Provides a name that will be used to identify your widget board. This is a required field. You are currently on a text field. To enter text in this field, type"* being announced. I think you can agree that this adds yet more context to the user and this is a good thing.

SemanticProperties.HeadingLevel

The `SemanticProperties.HeadingLevel` property allows you to mark an element as a heading to help organize the UI and make it easier for users to navigate. Some screen readers enable users to quickly jump between headings, thus providing a far more friendly navigation for those users that rely on screen readers. To give some context on the need for headings - when using VoiceOver on iOS you can swipe down or up to navigate between headings and then left or right to navigate between the items under the heading, otherwise it could be an arduous task for a user to navigate between all items in the UI in order to reach the item they need. Headings have a level from 1 to 9 and are represented by the `SemanticHeadingLevel` enumeration.

Setting SemanticProperties from Code

All of our examples have set the properties inside XAML, but we can also set them in C#; in fact, it looks rather different, so let's see how to use them. If you recall in Chapter 7 we introduced the `Placeholder` control which allows the user to interact with it. We are using a `Label` which does not typically support user interaction, and therefore, the screen reader will not inform the user that they can interact with it. Thankfully we already know how to fix this; let's open up the *Placeholder.cs* file and make the following changes in **bold**:

```
public Placeholder()
{
    Content = new Label
    {
        Text = "Tap to add widget",
        FontAttributes = FontAttributes.Italic,
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    };

    SemanticProperties.SetDescription(
        Content,
        "Tap to add a widget");
    SemanticProperties.SetHint(
        Content,
        "Allows you to choose a widget that can be added to the
        board at this location.");
}
```

You will notice how this approach looks rather different to setting the properties in XAML.

SemanticScreenReader

So far we have added helpful property values that the screen reader will use; this is great for on-screen control, but what if you want to provide more context for when an action was triggered? .NET MAUI provides the `SemanticScreenReader` that enables you to instruct a screen reader to announce some text to the user. This can work especially well if you wish to present instructions to a user or to prompt them if they have paused their interaction.

The `SemanticScreenReader` provides a static `Announce` method to perform the announcements; it also provides a `Default` instance. I personally like to make use of the scenarios where .NET MAUI provides you with a `Current` or a `Default` instance and register this with the app builder to make full use of the dependency injection support. To do this, write the following line of code in your `MauiProgram.cs` file:

```
builder.Services.AddSingleton(SemanticScreenReader.Default);
```

With the screen reader registered, you can announce that the new board was created successfully once the user has tapped on the Save button. You need to open the `BoardDetailsPageViewModel.cs` file and make the following changes.

Add the read-only field.

```
private readonly ISemanticScreenReader semanticScreenReader;
```

Assign a value in your constructor, just applying the **bold code** to your existing content.

```
public BoardDetailsPageViewModel(ISemanticScreenReader
semanticScreenReader)
{
    this.semanticScreenReader = semanticScreenReader;
    SaveCommand = new Command(
        () => Save(),
        () => !string.IsNullOrWhiteSpace(BoardName));
}
```

Call Announce in your Save method, just applying the **bold code** to your existing content.

```
private async void Save()
{
    var board = new FixedBoard
    {
        Name = BoardName,
        NumberOfColumns = NumberOfColumns,
        NumberOfRows = NumberOfRows
    };

    semanticScreenReader.Announce($"A new board with the name
    {BoardName} was created successfully.");

    await Shell.Current.GoToAsync(
        RouteNames.FixedBoard,
        new Dictionary<string, object>
        {
            { "Board", board }
        });
}
```

If you run your application and save a new board called "My work board," you will observe that the screen reader will announce *"A new board with the name My work board was created successfully."* This gives the user some valuable audible feedback. If you expect the save process to take some time, you can also perform an announcement at the start of the process to keep the user informed.

AutomationProperties

`AutomationProperties` are the old Xamarin.Forms way of exposing information to the screen readers on each platform. I won't cover all of the options because some have been replaced by the `SemanticProperties` section that you just learned about. In fact, I would strongly recommend that you always look at `SemanticProperties` before considering using the `AutomationProperties` class. The following are the important ones that provide a different set of functionality.

AutomationProperties.ExcludedWithChildren

The `AutomationProperties.ExcludeWithChildren` property allows developers to exclude the element supplied and all its children from the accessibility tree. Setting this property to `true` will exclude the element and all of its children from the accessibility tree.

AutomationProperties.IsInAccessibleTree

The `AutomationProperties.IsInAccessibleTree` property allows developers to decide whether the element is visible to screen readers. A common scenario for this feature is to hide controls such as `Label` or `Image` controls that serve a purely decorative purpose (e.g., a background image). Setting this property to `true` will exclude the element from the accessibility tree.

Suitable Contrast

WCAG states in guideline *1.4.3 Contrast (Minimum) – Level AA* that the visual presentation of text and images of text has a contrast ratio of at least 4.5:1, except for the following:

- Large Text: Large-scale text and images of large-scale text have a contrast ratio of at least 3:1.
- Incidental: Text or images of text that are part of an inactive user interface component, that are pure decoration, that are not visible to anyone, or that are part of a picture that contains significant other visual content have no contrast requirement.
- Logotypes: Text that is part of a logo or brand name has no contrast requirement.

This all boils down to calculating the difference between the lighter and darker colors in your application when displaying text. If that contrast ratio is 4.5:1 or higher, it's suitable. Let's look at how this is calculated:

```
(L1 + 0.05) / (L2 + 0.05)
```

where L1 is the relative luminance of the lighter color and L2 is the relative luminance of the darker color. Relative luminance is defined as *the relative brightness of any point in a color space, normalized to 0 for darkest black and 1 for lightest white*. Relative luminance can be further calculated as

```
For the sRGB colorspace, the relative luminance of a color is
defined as L = 0.2126 * R + 0.7152 * G + 0.0722 * B where R, G
and B are defined as:
if RsRGB <= 0.03928 then R = RsRGB/12.92 else R =
((RsRGB+0.055)/1.055) ^ 2.4
```

```
if GsRGB <= 0.03928 then G = GsRGB/12.92 else G =
((GsRGB+0.055)/1.055) ^ 2.4
if BsRGB <= 0.03928 then B = BsRGB/12.92 else B =
((BsRGB+0.055)/1.055) ^ 2.4
and RsRGB, GsRGB, and BsRGB are defined as:
RsRGB = R8bit/255
GsRGB = G8bit/255
BsRGB = B8bit/255
The "^" character is the exponentiation operator.
```

These formulas are taken from www.w3.org/TR/WCAG21/#dfn-relative-luminance. Let's turn this into some C# to make it a little easier to follow and something that you can use to test your color choices.

```
private static double GetContrastRatio(Color lighterColor,
Color darkerColor)
{
    var l1 = GetRelativeLuminance(lighterColor);
    var l2 = GetRelativeLuminance(darkerColor);
    return (l1 + 0.05) / (l2 + 0.05);
}
private static double GetRelativeLuminance(Color color)
{
    var r = GetRelativeComponent(color.Red);
    var g = GetRelativeComponent(color.Green);
    var b = GetRelativeComponent(color.Blue);
    return
        0.2126 * r +
        0.7152 * g +
        0.0722 * b;
}
```

```
private static double GetRelativeComponent(float component)
{
    if (component <= 0.03928)
    {
        return component / 12.92;
    }
    return Math.Pow(((component + 0.055) / 1.055), 2.4);
}
```

If you take a look at the colors you are using for your text controls and the background colors, you can work out whether you need to improve on the contrast ratio. You can see by checking in your `Styles.xaml` file that your Label control uses `Gray900` for the text color. Checking in the `Colors.xaml` file, you can see that this `Gray900` color has a value of `#212121`. Therefore, you can use your methods to calculate the contrast ratio with

```
GetContrastRatio(Colors.White, Color.FromArgb("#212121");
```

This gives you a contrast ratio of 16.10:1, which means this is providing a very good contrast ratio. The best possible contrast is black on white, which gives a contrast ratio of 21:1. Therefore, you do not need to make any changes to your color scheme, which shows that .NET MAUI ships with default color options that are suitable for building accessible applications. In fact, I have it on good authority that the .NET MAUI templates undergo audits to ensure that they are accessible; therefore, they provide an excellent set of examples to follow.

Dynamic Text Sizing

WCAG states in guideline *1.4.4 Resize text – Level AA* that except for captions and images of text, text can be resized without assistive technology up to 200% without loss of content or functionality.

This guideline mainly focuses on highlighting the fact that there is still a large percentage of users that do not rely on accessibility features such as screen readers or screen magnification when they could benefit from them. The guideline further states that, as a developer, you should provide the ability to scale the text in your application up to 200% without relying on the operating system to perform the scaling.

In this section, I am not going to focus on adding that specific feature; however, I will be discussing some approaches that will aid this feature as well as using the assistive technology options.

Avoiding Fixed Sizes

Wherever possible, you want to avoid setting the `WidthRequest` and `HeightRequest` properties for any control that can contain text.

Imagine you set `WidthRequest="200"` and `HeightRequest="30"` on the `Label` controls in your `BoardDetailsPage.xaml` file. What you would initially see is that the text fits nicely using the standard font scaling options. Figure 8-1 shows your application with fixed size controls and a small font size.

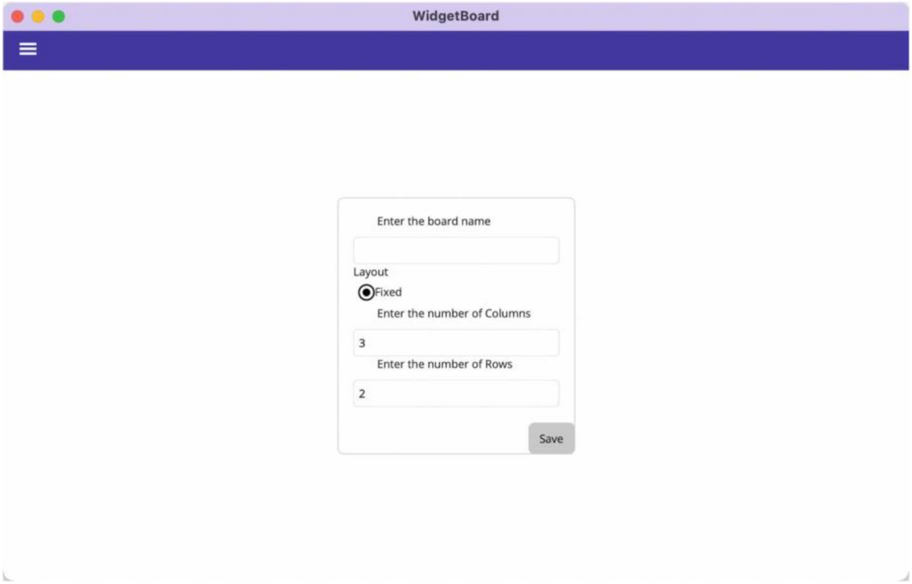

Figure 8-1. *Your application with fixed sizing and a small font size*

However, if you up the scaling to 200%, you will see a rather unpleasant screen. Figure 8-2 shows your application with fixed size controls and a large font size, highlighting that the text becomes clipped and unreadable.

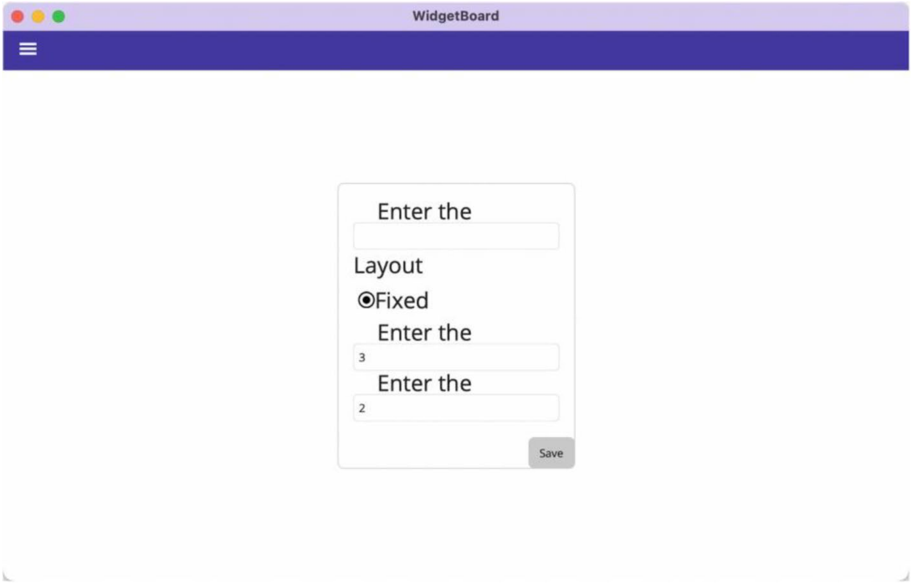

Figure 8-2. *Your application with fixed sizing and a large font size*

It actually appears that your initial changes without the `WidthRequest` and `HeightRequest` values on the `Label` controls gives the best experience. Figure 8-3 shows your application responding to font size changes when control sizes are not fixed.

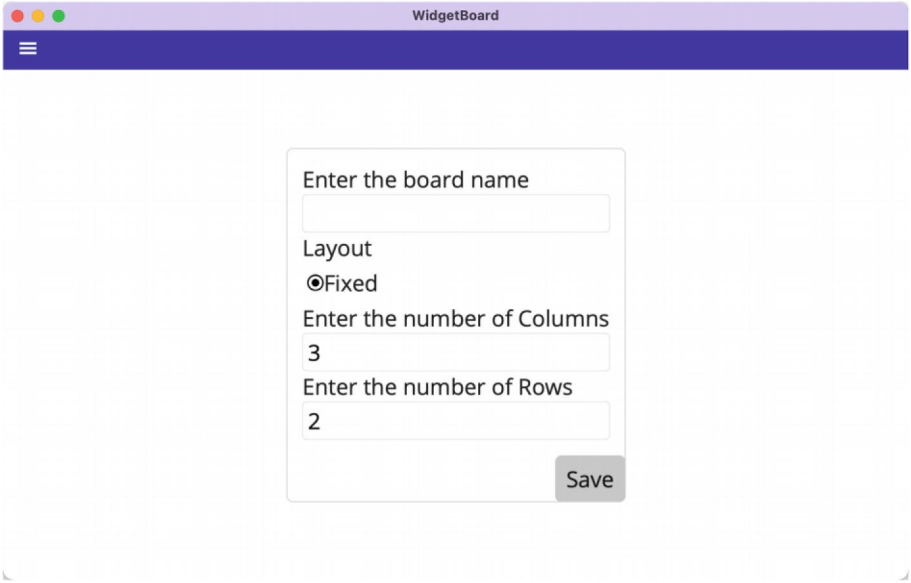

Figure 8-3. *Your application showing responsiveness to font scaling*

Preferring Minimum Sizing

Where possible, you should use `MinimumWidthRequest` and `MinimumHeightRequest` over `WidthRequest` and `HeightRequest`, respectively. This allows for controls to grow. There may be scenarios where a combination of Minimum and Maximum property values will give a good experience when scaling is introduced.

Font Auto Scaling

By default, all controls that render text in a .NET MAUI application have the `FontAutoScalingEnabled` property set to true. This means that the controls automatically scale their font size accordingly when the operating system's font scaling settings are changed.

There can be scenarios when disabling this feature can provide a more accessible experience. One example is in a wordsearch application I built. The application made the letters appear as big as possible, so any additional scaling by the operating system would result in parts of the text being cut off. I advise using this option sparingly.

Testing Your Application's Accessibility

Each platform supported by .NET MAUI has its own set of guidelines around testing for accessibility and even tools to aid that journey. In this section, you are going to take a brief look at what each platform provider offers.

Android

Google, much like each of the other platform providers, does recommend that you perform a manual test, such as turning on TalkBack and verifying that the user experience is as you have designed.

Google also offers some analysis tools to detect whether any accessibility guidelines are not being met. There is a good list provided by Google with a breakdown of the functionality provided by each tool at `https://developer.android.com/guide/topics/ui/accessibility/testing#analysis`.

iOS

Apple doesn't offer as much as Google on this front. There is the Accessibility Inspector, but it only focuses on allowing you to view the information that the screen reader will be provided. I don't feel this is as good as taking a dry run through your application with the VoiceOver assistant turned on. Further information on Apple's offering can be found at https://developer.apple.com/library/archive/technotes/TestingAccessibilityOfiOSApps/TestAccessibilityiniOSSimulatorwithAccessibilityInspector/TestAccessibilityiniOSSimulatorwithAccessibilityInspector.html.

macOS

Apple provides a little extra functionality when testing on macOS. It does provide the Accessibility Inspector as per iOS and well as the Accessibility Verifier. This tool allows you to run tests against your application to verify items like the accessibility description have been defined on all required elements. Further information on these features can be found at https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html.

Windows

Microsoft offers the biggest amount of options when it comes to testing the accessibility of your applications. The Windows Software Development Kit (SDK) provides several tools such as the ability to inspect an application and view all related properties plus automation tests that verify the state of accessibility. All details of the tools can be found at https://docs.microsoft.com/windows/apps/design/accessibility/accessibility-testing.

Useful Resources

Accessibility Checklist

The following checklist is provided by Microsoft on their documentation site at `https://docs.microsoft.com/dotnet/maui/fundamentals/accessibility#accessibility-checklist`. I haven't added to it or reworded because I believe it provides an excellent breakdown of the possible ways to provide accessible support.

Follow these tips to ensure that your .NET MAUI apps are accessible to the widest audience possible:

- Ensure your app is perceivable, operable, understandable, and robust for all by following the Web Content Accessibility Guidelines (WCAG). WCAG is the global accessibility standard and legal benchmark for web and mobile. For more information, see Web Content Accessibility Guidelines (WCAG) Overview.
- Make sure the user interface is self-describing. Test that all the elements of your user interface are screen reader accessible. Add descriptive text and hints when necessary.
- Ensure that images and icons have alternate text descriptions.
- Support large fonts and high contrast. Avoid hard-coding control dimensions, and instead prefer layouts that resize to accommodate larger font sizes. Test color schemes in high-contrast mode to ensure they are readable.

- Design the visual tree with navigation in mind. Use appropriate layout controls so that navigating between controls using alternate input methods follows the same logical flow as using touch. In addition, exclude unnecessary elements from screen readers (e.g., decorative images or labels for fields that are already accessible).
- Don't rely on audio or color cues alone. Avoid situations where the sole indication of progress, completion, or some other state is a sound or color change. Either design the user interface to include clear visual cues, with sound and color for reinforcement only, or add specific accessibility indicators. When choosing colors, try to avoid a palette that is hard to distinguish for users with color blindness.
- Provide captions for video content and a readable script for audio content. It's also helpful to provide controls that adjust the speed of audio or video content, and ensure that volume and transport controls are easy to find and use.
- Localize your accessibility descriptions when the app supports multiple languages.
- Test the accessibility features of your app on each platform it targets. For more information, see Testing Your Application's Accessibility.

A Guide for Making Apps Accessible

Another set of great resource is the Appt website `https://appt.org/en/`. To quote their website:

> *"Appt.org empowers developers and organizations to build accessible apps for everyone. On this website you will find current statistics about the use of accessibility features on mobile phones, code documentation for iOS, Android and other platforms, official guidelines, in-depth articles and tips."*

They also provide a set of resources specifically for building accessible .NET MAUI applications at `https://appt.org/en/docs/net-maui/samples`.

Summary

In this chapter, you have

- Gained an understanding of what accessibility is
- Learned why it is important to build inclusive applications
- Looked at how you can make use of .NET MAUI functionality
- Considered other scenarios and how to support them
- Looked over some testing options to support your journey to building accessible applications

In the next chapter, you will

- Add a widget to a board
- Explore the different options available when showing an overlay
- Explore how you can define styling information for your application
- Learn how to handle devices running in light and dark modes

- Learn how to apply triggers to enhance your UI
- Explore how to animate parts of your application
- Explore what happens when you combine triggers and animations together

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch08`.

Extra Assignment

Take one of your favorite applications that you are completely familiar with because you know the layout and how to use it. Then proceed to

- Turn on the screen reading assistant on your phone.
- Try to navigate your way around this application.
- Better still, try to impact your vision with a blindfold or remove any glasses if you use them. Try to rely entirely on the screen reader.
- Perhaps try the same but modify the device font scaling and see if the application is able to handle increases in text size, or if it even allows this option.

The objective is to gain a sense of the experience users with limited vision have when using the same application. Take notes on how well applications do things and how poorly they do other things. This can be a really great learning exercise for you all!

CHAPTER 9

Advanced UI Concepts

Abstract

In this chapter, you will provide the user of your application with the ability to add a widget to the boards they create through the use of an overlay. You will further enhance this overlay by defining common styling techniques and handling the differences between light and dark mode devices.

You will then take a journey into discovering how you can build an application that feels natural and organic to your human user base. Finally, you will look at how you can keep the animations driving the organic look and feel cleanly separated from your business logic code.

As we covered in Chapter 7, there are two common approaches to extending functionality: inheritance and composition. Chapter 7 focused on how to utilize inheritance in how we built our BoardLayout control; this chapter will focus on the composition approach through highlighting all of the pieces that .NET MAUI offers to make this an easy task. The two key offerings that we will be covering are Triggers and Behaviors – these each have their own sections in this chapter.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_9

Adding the Ability to Add a Widget to a Board

In Chapter 7, you created your own `BoardLayout` and the associated `FixedLayoutManager` that enabled you to show a board and added in the ability to handle interaction events by the user. In this section, you are going to expand on that to handle the user tapping on a widget `Placeholder` and letting the user choose a widget to add to the board.

Possible Ways of Achieving Your Goal

There are several ways you can go about adding in this piece of functionality. Some are better suited to different scenarios, and some simply come down to a personal preference. I encourage you to understand your goal before you embark on this journey of working out which option will best suit your needs. If you only wish to report a message to the user or capture a choice or even a single piece of input, then you can utilize some underlying functionality provided by .NET MAUI. The `Page` class provides the ability to do each of the three items discussed; it doesn't solve your needs, but it really does have value in many applications. The Microsoft documentation provides a good set of reference examples on how to use these options at https://learn.microsoft.com/dotnet/maui/user-interface/pop-ups.

Let's discuss some of the options that do solve your needs and then make a decision on which you feel is the best candidate for your application.

Showing a Modal Page

So far in this book you have only considered how `Shell` offers the ability to navigate between `ContentPages`. This is the default and most common scenario. There can be times when you wish to show a page that is blocking and will require the user to engage with it to return to the previous page. This type of page or display is referred to as *modal*. The scenario of showing something to the user and requiring them to engage with it could be a perfect scenario.

In order to enable this functionality in .NET MAUI, you need to set the `Shell.PresentationMode` property on the `ContentPage` that you wish to display. For example:

```
<ContentPage ...
             Shell.PresentationMode="Modal">
    ...
</ContentPage>
```

You can then call the `Shell.Current.GoToAsync` method with the routing options configured for this page and it will be presented modally instead of being navigated to.

Pro

- Keeps specific code contained

Con

- Complicates flow of code when handling a return result as we learned in Chapter 6 when we used a `TaskCompletionSource`

Overlaying a View

Sometimes the most straightforward way to achieve this approach is to add another view to your page and programmatically change its visibility to give the impression you have a modal page displaying.

Pro

- Reduces effort of page creation

Con

- Requires specific code in calling view/view model

Showing a Popup

There is currently no explicit support in .NET MAUI for displaying popups; however, the functionality does exist on each of the platforms that .NET MAUI runs on. You can go to the lengths of implementing your own ability to display a popup, but it would be rather involved. Instead, the .NET MAUI Community Toolkit provides a `Popup` class that makes it straightforward for you to display a popup in your application.

Pros

- Keeps specific code contained
- Provides easy return result handling

Con

- Brings in an extra dependency

For further reading on how to use the toolkit and its `Popup` class, please refer to the documentation at `https://learn.microsoft.com/dotnet/communitytoolkit/maui/views/popup`.

The Chosen Approach

Given the pros and cons outlined above, you might guess that you will be using the `Popup` class. Nope. Let's use the overlaying-a-view approach. This is mainly because it will help to expose you to more .NET MAUI-specific concepts that I believe will be extremely valuable in building applications. However, for your own work, use the approach that best fits your scenario. I would like to emphasize that each of the above options will achieve the results needed. In fact, there could well be more options that I haven't covered, and if you find one, I would love to hear about it.

Adding Your Overlay View

You need to add a view to your `FixedBoardPage.xaml` file that will present the option to the user to add a new widget to the board. Let's open that file and add the following code inside the `Grid` and below the `</layouts:BoardLayout>` line:

```
<BoxView
    BackgroundColor="Black"
    Opacity="0.5"
    IsVisible="{Binding IsAddingWidget}" />
<Border
    IsVisible="{Binding IsAddingWidget}"
    HorizontalOptions="Center"
    VerticalOptions="Center"
    Padding="10">
    <VerticalStackLayout>
        <Label
            Text="Add widget"
            FontSize="20" />
        <Label
            Text="Widget" />
        <Picker
            ItemsSource="{Binding AvailableWidgets}"
            SelectedItem="{Binding SelectedWidget}"
            SemanticProperties.Description="{Binding Text,
            Source={x:Reference SelectTheWidgetLabel}}"
            SemanticProperties.Hint="Picker containing the
            possible widget types that can be added to the
            board. This is a required field." />
```

```
        <Label
            Text="Preview" />
        <ContentView
            WidthRequest="250"
            HeightRequest="250" />
        <Button
            Text="Add widget"
            Command="{Binding AddWidgetCommand}"
            SemanticProperties.Hint="Adds the selected widget
            to the board. Requires the 'Select the widget'
            field to be set." />
    </VerticalStackLayout>
</Border>
```

The code addition results in two new controls added to the parent `Grid`'s children collection: a `BoxView` and a `Border`. The `BoxView` is added to provide a semi-transparent overlay on top of the rest of the application, and the `Border` presents the content for selecting a new widget. Adding them after the `BoardLayout` means it will be rendered on top of the `BoardLayout`. This ordering is referred to as Z-index, and in the majority of .NET MAUI applications, layouts are determined by the order in which the children are added to their parent. This means that the later the controls are added, the higher they will appear visually. You can modify this default behavior by using the `ZIndex` property where the higher the value, the higher they will appear visually. With this knowledge, you can add a binding between the `IsVisible` property of your new controls and a property on your view model, so your view model can control whether you are adding a widget to the board.

Let's update your view model.

Updating Your View Model

Since you turned on compiled bindings in a previous chapter, you will now see that your code will not compile because you have not defined the properties you are binding to. So open the `FixedBoardPageViewModel.cs` file and make the following additions.

Add the new properties and associated backing fields into your `FixedBoardPageViewModel` class.

```
private int addingPosition;
private string? selectedWidget;
private bool isAddingWidget;
private readonly WidgetFactory widgetFactory;

public IList<string> AvailableWidgets => widgetFactory.
AvailableWidgets;

public ICommand AddWidgetCommand { get; }
public ICommand AddNewWidgetCommand { get; }

public bool IsAddingWidget
{
    get => isAddingWidget;
    set => SetProperty(ref isAddingWidget, value);
}

public string? SelectedWidget
{
    get => selectedWidget;
    set => SetProperty(ref selectedWidget, value);
}
```

Update the constructor with the new `WidgetFactory` dependency and set the new commands that you have added; changes are in **bold**.

```
public FixedBoardPageViewModel(
    WidgetTemplateSelector widgetTemplateSelector,
    WidgetFactory widgetFactory)
{
    WidgetTemplateSelector = widgetTemplateSelector;
    this.widgetFactory = widgetFactory;
    Widgets = new ObservableCollection<IWidgetViewModel>();
    AddWidgetCommand = new Command(OnAddWidget);
    AddNewWidgetCommand = new Command<int>(index =>
    {
        IsAddingWidget = true;
        addingPosition = index;
    });
}
```

In the previous code section, you set the `IsAddingWidget` property to `true` in order to show the overlay view and you also keep a record of the index variable, which is the `Position` property from the `Placeholder` that was tapped.

Provide the method implementation for the `AddWidgetCommand`.

```
private void OnAddWidget()
{
    if (SelectedWidget is null)
    {
        return;
    }

    var widgetViewModel = widgetFactory.CreateWidgetViewModel
    (SelectedWidget);
```

```
    if (widgetViewModel is not null)
    {
        widgetViewModel.Position = addingPosition;
        Widgets.Add(widgetViewModel);
    }

    IsAddingWidget = false;
}
```

Hopefully the majority of what you just added should feel familiar. The part that most likely doesn't is the final `OnAddWidget` method. Let's take a deeper look at this implementation.

The `SelectedWidget` property is bound to your `Picker` in the view. You do some initial input validation to make sure that the user has chosen a type of widget to add; otherwise, you return out of the method.

Next, you use the new dependency (`widgetFactory`) to create a view model for you.

Then you set its `Position` based on which placeholder was tapped initially.

Then you add your newly created `widgetViewModel` to the collection of `Widgets` so that it can update the UI.

Finally, you set the `IsAddingWidget` property to false in order to hide the overlay view again.

Showing the Overlay View

Now you can add the ability to programmatically show the `Border` that allows your users to pick a widget and add it to the board. You already provided a large amount of this functionality inside your `Placeholder` and `FixedLayoutManager` classes, so you just need to hook up your view model to this functionality. You have also just set the groundwork in your view model, so let's hook the components up. Open the `FixedBoardPage.xaml` file again and add the following **bold** line:

```
<layouts:BoardLayout
    ItemsSource="{Binding Widgets}"
    ItemTemplateSelector="{Binding WidgetTemplateSelector}">
    <layouts:BoardLayout.LayoutManager>
        <layouts:FixedLayoutManager
            NumberOfColumns="{Binding NumberOfColumns}"
            NumberOfRows="{Binding NumberOfRows}"
            PlaceholderTappedCommand="{Binding AddNewWidget
            Command}" />
    </layouts:BoardLayout.LayoutManager>
</layouts:BoardLayout>
```

If you build and run your application, you can see that once you have created a board, you can now tap or click on the `Placeholder` and observe that your overlay displays. You will notice that there is no background to your overlay, though, so it is really difficult for a user to understand what to do. You can just set the `BackgroundColor` of your `Border` control; however, this can lead to a number of issues. For example, if you fixed the `BackgroundColor` to white and a user switches on dark mode on their device, they would have a rather unpleasant experience. Figure 9-1 shows how the application currently looks and highlights the issue.

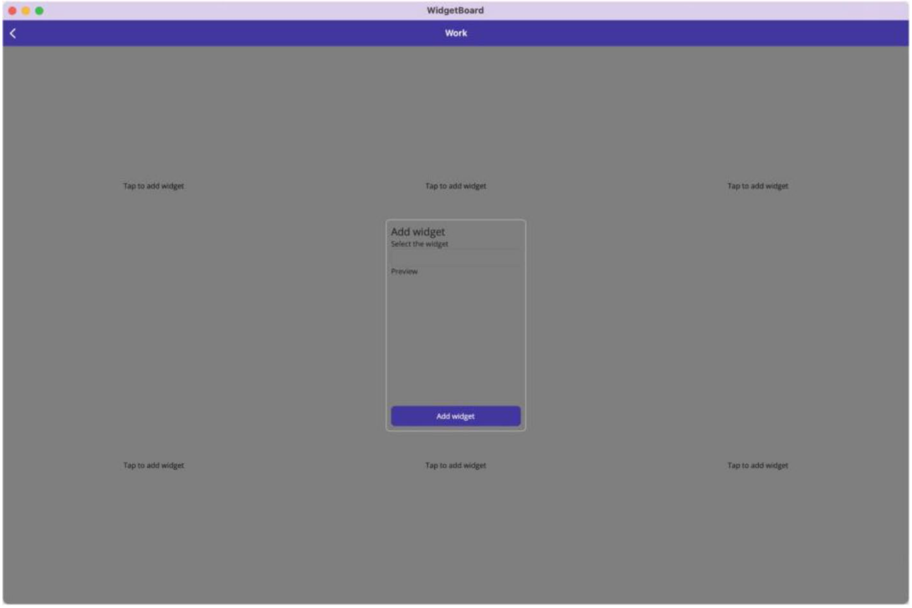

Figure 9-1. *The application showing the overlay with a poor user experience*

Let's look at how .NET MAUI provides the ability to style your applications, which includes supporting light and dark modes.

Styling

.NET MAUI provides the ability to style your applications. Styling in .NET MAUI offers many advantages:

- Central definition of look and feel
- Less verbosity in your XAML/code

- Style inheritance

  Styles in .NET MAUI can be defined at many different levels, and where they are defined is extremely important when understanding what impact they will have. The two key distinctions between where they are defined can be considered as

- Globally: These styles are added to the application's resources. You can see an example of this if you open the `App.xaml` file. The line in bold shows that another file (`Styles.xaml`) containing the styles is loaded into the `Application.Resources` property. These styles apply to all controls in the application unless otherwise explicitly overridden.

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

- Locally: These styles are added to a view or page resources property. Styles defined in this way will apply to all controls that are children of the view or page they are defined in.

Examining the Default Styles

You can view this file under `Resources/Styles.xaml`. Let's take a look at the style for `Border` in this file:

```
<Style TargetType="Border">
    <Setter Property="Stroke" Value="{AppThemeBinding
    Light={StaticResource Gray200}, Dark={StaticResource
    Gray500}}" />
    <Setter Property="StrokeShape" Value="Rectangle"/>
    <Setter Property="StrokeThickness" Value="1"/>
</Style>
```

The XAML syntax used to define a style looks rather different to the XAML you have written so far. Let's break it down to gain a better understanding of what it all means.

TargetType

To start, when defining a `Style`, you must define the `TargetType`. This property defines which type of control the style definition targets and therefore applies to. Defining a `Style` with only the `TargetType` property set will apply to all controls of that type within the scope it is defined. This is referred to as *implicit styling*.

If you wish to explicitly style a control, you can also add the `x:Key` property. This is referred to as *explicit styling*. You are then required to set the `Style` property on any control that wishes to use this explicit style that you have created. You will be creating an explicit style in the "Creating a Style" section following shortly.

ApplyToDerivedTypes

By default, styles created explicitly apply to the type defined in the `TargetType` property I just covered. If you wish to allow derived classes to also inherit this style, you need to set the `ApplyToDerivedTypes` property to `true`. If you have a `CustomBorder` that inherits from `Border` and you had the following `Style` defined:

```
<Style TargetType="Border" ApplyToDerivedTypes="True">
    <Setter Property="Stroke" Value="{AppThemeBinding
    Light={StaticResource Gray200}, Dark={StaticResource
    Gray500}}" />
    <Setter Property="StrokeShape" Value="Rectangle"/>
    <Setter Property="StrokeThickness" Value="1"/>
</Style>
```

then the `Style` will also be applied to your `CustomBorder` control. If `ApplyToDerivedTypes` was set to `false`, it would not be applied.

Setter

This is the part that looks and feels quite a bit different to the previous XAML you have written. Since you are not creating controls but defining how they will look, you must follow this syntax. Let's look at the following example:

```
<Style TargetType="Label">
    <Setter Property="TextColor" Value="Black" />
</Style>
```

The above is not a style you would include in an application; however, as an example it allows you to say

The `Style` for `Label` controls will set the `TextColor` property to `Black`.

Now that you have had a look at some of the key concepts that make up a style in .NET MAUI, let's create your own style for your overlay.

Creating a Style

Let's view this in action by adding the following to the `Styles.xaml` file. Add this just below the existing `<Style TargetType="Border">` entry.

```
<Style TargetType="Border" x:Key="OverlayBorderStyle">
    <Setter Property="BackgroundColor" Value="White" />
    <Setter Property="Stroke" Value="{AppThemeBinding
    Light={StaticResource Gray200}, Dark={StaticResource
    Gray500}}" />
    <Setter Property="StrokeShape" Value="Rectangle"/>
    <Setter Property="StrokeThickness" Value="1"/>
</Style>
```

The above looks very similar to the default `Border` style already defined with the addition of the `BackgroundColor` setter.

It is also worth noting that you only need to set the values that you wish to change from the implicit style. Therefore, your explicit style can be reduced down to

```
<Style TargetType="Border" x:Key="OverlayBorderStyle">
    <Setter Property="BackgroundColor" Value="White" />
</Style>
```

The `Stroke`, `StrokeShape`, and `StrokeThickness` properties will all be inherited from the implicit global style. This provides yet another great way to reduce the amount of code you need to write.

Now you can use this style in your application. Open the `FixedBoardPage.xaml` file and add the following line to your `Border` element (change in **bold**):

```
<Border
    IsVisible="{Binding IsAddingWidget}"
    HorizontalOptions="Center"
    VerticalOptions="Center"
    Padding="10"
    Style="{StaticResource OverlayBorderStyle}">
```

This will result in your overlay looking far better to the user now because it is no longer transparent. Also, consider moving the `HorizontalOptions`, `VerticalOptions`, and `Padding` properties over to the style definition. Figure 9-2 shows how much better the overlay now looks.

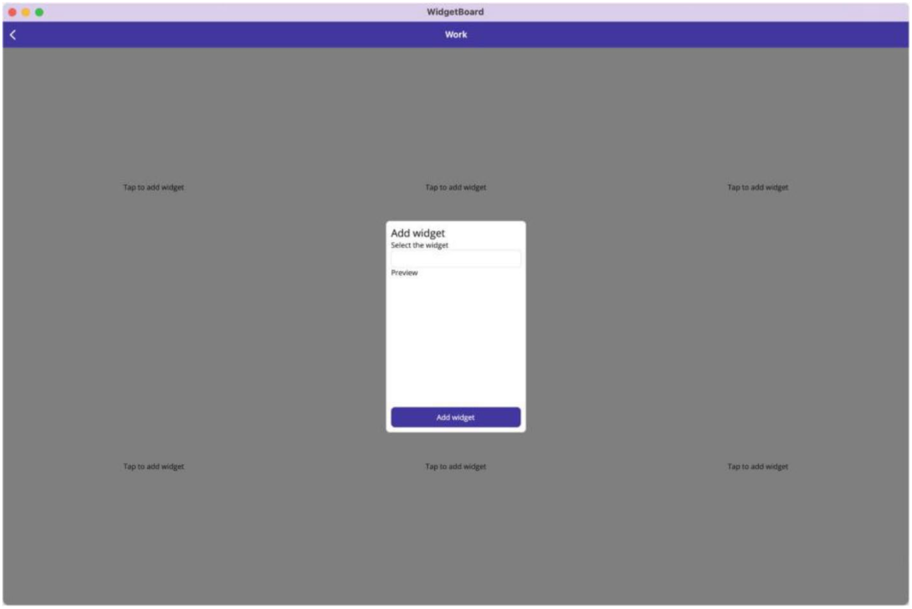

***Figure 9-2.** The overlay with a much clearer background*

What you have done here is considered bad practice, though! You have hard-coded the `BackgroundColor` of your `Border` control in the style definition so your application will look great on a device running in light mode. However, as soon as the user switches to dark mode, they will have a glaring white border showing.

The repercussions of using fixed values can include text or content disappearing entirely from the application. Imagine that the text color switches to white in dark mode, with you having hard-coded to a white background of the overlay view, so the user would see no text on screen. This would result in a terrible user experience.

.NET MAUI provides the ability to handle the different modes that a device can run under.

AppThemeBinding

This is an extremely valuable concept. It allows you to define different values based on whether the device your application is running on is set to light or dark mode. Taking the example of the `OverlayBorderStyle` you previously created, you can modify the `Setter` for `BackgroundColor` to

```
<Setter Property="BackgroundColor" Value="{AppThemeBinding
Light={StaticResource White}, Dark={StaticResource Black}}" />
```

Now if a user is running in dark mode, the border overlay will be black and the text will be visible.

You only need to apply `AppThemeBinding` to properties that require a visual distinction between light and dark modes. This typically applies to all `Brush`/`Color` properties; however, you could conceivably decide to change the `StrokeThickness` of your `Border` control, for example.

Further Reading

It is worth noting that this book is limited to covering the styling options in XAML. However, .NET MAUI does provide support for CSS-based style sheets. Go to `https://docs.microsoft.com/dotnet/maui/user-interface/styles/css`.

Triggers

.NET MAUI provides a concept called triggers. They enable you to further enhance how your views react to changes in the view model. You are given the ability to define actions that can modify the appearance of the UI based on event or data changes. Triggers provide us with another way of changing the visibility of our border overlay for adding a new widget. The initial work will appear more verbose in the short term, but do bear with me – it will result in a much better outcome!

There are a number of different types of triggers that can be attached to a control, each with a varying level of functionality. You will take a brief look at them and then dig into the one that you need for your scenario.

- `Trigger`: A `Trigger` represents a trigger that applies property values, or performs actions, when the specified property meets a specified condition.
- `DataTrigger`: A `DataTrigger` represents a trigger that applies property values, or performs actions, when the bound data meets a specified condition. The Binding markup extension is used to monitor for the specified condition.
- `EventTrigger`: An `EventTrigger` represents a trigger that applies a set of actions in response to an event. Unlike `Trigger`, `EventTrigger` has no concept of termination of state, so the actions will not be undone once the condition that raised the event is no longer true.

- `MultiTrigger`: A `MultiTrigger` represents a trigger that applies property values, or performs actions, when a set of conditions are satisfied. All the conditions must be true before the Setter objects are applied.

Creating a DataTrigger

In this chapter, you have added your overlay `Border` control and are currently changing its visibility through a binding direct to the `IsVisible` property. You can write this differently with a `DataTrigger`. Let's open the `FixedBoardPage.xaml` file and modify the `Border` control to the following:

```
<Border
    IsVisible="False"
    HorizontalOptions="Center"
    VerticalOptions="Center"
    Padding="10"
    Style="{StaticResource OverlayBorderStyle}">
    <Border.Triggers>
        <DataTrigger
            TargetType="Border"
            Binding="{Binding IsAddingWidget}"
            Value="True">
            <Setter
                Property="IsVisible"
                Value="True" />
        </DataTrigger>
    </Border.Triggers>
```

Notice that the syntax for a `Trigger` is very similar to a `Style`. You will also notice that it looks a lot more verbose than your original simple binding approach. If you simply want to control the `IsVisible` property of a control, a trigger is overkill, in my opinion. You will not be ending here, though, so bear with me. First, let's break down what you have added and then look at how you can enhance it.

First, you modify the `IsVisible` property binding to `false`. This is the initial state of the visibility of your view.

```
IsVisible="False"
```

Next, you add the `DataTrigger` to the `Border.Triggers` property.

```
<DataTrigger
    TargetType="Border"
    Binding="{Binding IsAddingWidget}"
    Value="True">
```

Much like with styles, you define the type of control the `DataTrigger` applies to. You also set the `Binding` property to bind to the `IsAddingWidget` property on your view model. Finally, you set the `Value` property to `true`. This all means that when the `IsAddingWidget` property value is set to `true`, the contents of the `DataTrigger` will be applied.

This leads you onto the final change, which is the setter.

```
<Setter
    Property="IsVisible"
    Value="True" />
```

To repeat myself, all of this is rather verbose until you consider that you can define actions that can be performed when your state is entered/exited.

EnterActions and ExitActions

As an alternative to simply defining values for properties to be set when the `IsAddingWidget` property value becomes true, like in your previous example, you can define actions that will be performed when the value enters or exits a specific state. What exactly does this mean? Let's take a look at an example. You can rewrite the trigger usage from the previous example as

```
<DataTrigger
    TargetType="Border"
    Binding="{Binding IsAddingWidget}"
    Value="True">
    <DataTrigger.EnterActions>
        <!--action to perform-->
    </DataTrigger.EnterActions>
    <DataTrigger.ExitActions>
        <!--action to perform-->
    </DataTrigger.ExitActions>
</DataTrigger>
```

Given the above, you can state the following:

When the property (IsAddingWidget) in the Binding enters the state defined in Value (True), the EnterActions will be performed.

When the property (IsAddingWidget) in the Binding exits the state defined in Value (False), the ExitActions will be performed.

You need to define an action to be performed for these scenarios now.

Creating a TriggerAction

.NET MAUI provides the TriggerAction<T> base class that allows you to define an action that will be performed in the enter or exit scenario. This enables you to build a more complex behavior that can be performed when a value changes. When creating a trigger action, you can use the base class TriggerAction<T> provided by .NET MAUI, and then you need to override the Invoke method. It is this method that defines what action will be performed when the value changes. Let's create your own action that you can use.

Creating ShowOverlayTriggerAction

First, you need to find a place to locate this action. Create a new folder in the root project called `Triggers` and then add a new class file called `ShowOverlayTriggerAction.cs`. Then you can add the following code:

```
namespace WidgetBoard.Triggers;

public class ShowOverlayTriggerAction :
TriggerAction<VisualElement>
{
    public bool ShowOverlay { get; set; }
    protected override void Invoke(VisualElement sender)
    {
        sender.IsVisible = ShowOverlay;
    }
}
```

This code doesn't do too much right now. It will just change the `IsVisible` property of the control it is attached to when the value changes.

Now you need to attach it to your `AddWidgetFrame` control.

Using ShowOverlayTriggerAction

You can now add in the action to perform sections that you left when first adding a `DataTrigger` to your control. Modify your code in the `FixedBoardPage.xaml` file, with the changes in **bold**.

```
<DataTrigger
    TargetType="Border"
    Binding="{Binding IsAddingWidget}"
    Value="True">
    <DataTrigger.EnterActions>
        <triggers:ShowOverlayTriggerAction
        ShowOverlay="True" />
```

```
    </DataTrigger.EnterActions>
    <DataTrigger.ExitActions>
        <triggers:ShowOverlayTriggerAction
        ShowOverlay="False" />
    </DataTrigger.ExitActions>
</DataTrigger>
```

This can now be interpreted as when the `IsAddingWidget` property value changes to `true`, a `ShowOverlayTriggerAction` will be invoked with `ShowOverlay` set to `true`. This will result in the `AddWidgetFrame` control becoming visible. Then, when the `IsAddingWidget` property value changes to `false`, a `ShowOverlayTriggerAction` will be invoked with `ShowOverlay` set to `false`. This will result in the `AddWidgetFrame` control becoming invisible.

It is also worth noting that you can define triggers in styles, meaning this type of functionality can be reused multiple times without having to duplicate the code.

Let's take a break from triggers for now to take a look at how you can animate controls in .NET MAUI. Then you will return to combine triggers and animations together to really show off the power of the action you just created.

Further Reading

You have only scratched the surface on the functionality that can be achieved with triggers. I recommend checking out the Microsoft documentation to see more ways triggers can be useful: `https://learn.microsoft.com/dotnet/maui/fundamentals/triggers`.

This feels like it could be a challenging topic to show off in printed form given the dynamic nature of an animation, but it is one of my favorite topics so I am going to show it off as best I can. Animations provide you with the building blocks to make your applications feel much more natural and organic.

.NET MAUI provides two main ways to perform an animation against any `VisualElement`. You will take a look at each approach and how some animations can be built using them.

Basic Animations

.NET MAUI ships with a set of prebuilt animations available via extension methods. These methods provide the ability to rotate, translate, scale, and fade a `VisualElement` over a period of time. Each of these methods has a *To* suffix, for example, `ScaleTo`. It is worth noting that each of the methods for animating is asynchronous and will therefore need to be awaited if you wish to know when they have finished. The full list of animation methods is as follows:

Method	Description
`FadeTo`	Animates the `Opacity` property of a `VisualElement`
`RelScaleTo`	Applies an animated incremental increase or decrease to the `Scale` property of a `VisualElement`
`RotateTo`	Animates the `Rotation` property of a `VisualElement`
`RelRotateTo`	Applies an animated incremental increase or decrease to the `Rotation` property of a `VisualElement`
`RotateXTo`	Animates the `RotationX` property of a `VisualElement`
`RotateYTo`	Animates the `RotationY` property of a `VisualElement`
`ScaleTo`	Animates the `Scale` property of a `VisualElement`
`ScaleXTo`	Animates the `ScaleX` property of a `VisualElement`
`ScaleYTo`	Animates the `ScaleY` property of a `VisualElement`
`TranslateTo`	Animates the `TranslationX` and `TranslationY` properties of a `VisualElement`

The overlay view you added in the previous section just shows immediately and disappears immediately based on the `IsVisible` binding you created. What if you animate your overlay to grow from nothing up to the required size? Don't worry about adding this code to your application just yet. You will look over some examples and then add it to Visual Studio in the "Combining Triggers and Animations" section. The main reason for not adding it immediately is because the animation's API relies on direct access to the view-related information, and this breaks the MVVM pattern. However, once you look over how to animate, you can take this learning and add it into your `ShowOverlayTriggerAction` implementation.

The code to animate a `VisualElement` is surprisingly small, as you can see in the following example:

```
AddWidgetFrame.Scale = 0;
await AddWidgetFrame.ScaleTo(1, 500);
```

First, you make sure that the `AddWidgetFrame` has a `Scale` of 0 and then you call `ScaleTo`, telling it to grow to a `Scale` of 1 (which is 100%) over a duration of 500 milliseconds.

All of the prebuilt animation methods apart from the ones that start with *Rel* perform the animation against the `VisualElement's` existing value (e.g., for `ScaleTo,` it will change from the existing `Scale` property value). This means that it is entirely possible that no animation will take place if both the existing property and the value provided to the method are the same.

Combining Basic Animations

It is entirely possible to combine the basic animations to provide much more complex animations. There are two main ways of achieving this.

Chaining Animations

You can chain animations together into a sequence. A common example here is to provide the appearance of a tile being flipped over and giving a 3D effect to the user. The key detail when chaining animations is that you await each animation method call to make sure that one animation has finished before the next one begins.

```
await frame.RotateXTo(90, 100);
frame.Content.IsVisible = tileViewModel.IsSelected;
await frame.RotateXTo(0, 100);
```

Concurrent Animations

In a similar way to chaining, you can perform multiple animations concurrently by simply not awaiting each method call or alternatively awaiting all of the calls.

```
AddWidgetFrame.Scale = 0;
AddWidgetFrame.IsVisible = true;
AddWidgetFrame.Opacity = 0;
await Task.WhenAll(
    AddWidgetFrame.FadeTo(1),
    AddWidgetFrame.ScaleTo(1, 500));
```

In fact, this animation looks like a very good contender for your actual implementation in the `ShowOverlayTriggerAction` implementation.

Cancelling Animations

Providing the ability to cancel an animation can be an extremely valuable feature for a user. Quite often in applications, and predominantly games, an animation will show when an action completes. Animations like this if blocking can become tiresome for users especially if the same animation repeats frequently. Therefore, a common pattern to follow is when the user taps on the control being animated, it cancels the animation.

If you wish to cancel an animation, you can call the `CancelAnimations` extension method on the `VisualElement` that you are animating.

```
AddWidgetFrame.CancelAnimations();
```

Easings

Animations in general will move mechanically as a computer changes a value over time. Easings allow you to move away from a linear update of those values in order to provide a much more organic and natural motion. .NET MAUI offers a whole host of prebuilt easings, plus there is even the ability to build your own if you really wish to do so. Let's take a look at the options that .NET MAUI provides out of the box:

Easing function	Description
`BounceIn`	Bounces the animation at the beginning
`BounceOut`	Bounces the animation at the end
`CubicIn`	Slowly accelerates the animation
`CubicInOut`	Accelerates the animation at the beginning and decelerates the animation at the end
`CubicOut`	Quickly decelerates the animation
`Linear`	Uses a constant velocity and is the default easing function
`SinIn`	Smoothly accelerates the animation
`SinInOut`	Smoothly accelerates the animation at the beginning and smoothly decelerates the animation at the end

(*continued*)

Easing function	Description
`SinOut`	Smoothly decelerates the animation
`SpringIn`	Causes the animation to very quickly accelerate toward the end
`SpringOut`	Causes the animation to quickly decelerate toward the end

As a general guide, an easing ending with the *In* suffix will start the animation slowly and speed up as it comes to a finish. An easing ending with the *Out* suffix will start off quickly and slow down toward the end.

Complex Animations

.NET MAUI provides the `Animation` class. This enables you to define complex animation sequences. In fact, the prebuilt animations that you covered in the "Basic Animations" section are built using this class inside the .NET MAUI code. Using this class, it is possible to animate any visual property of a `VisualElement`; for example, you can animate a change in `BackgroundColor` or `TextColor`.

The `Animation` class provides the ability to define simple animations through to really quite complex animations. Take a quick look at how the `ScaleTo` animation can be implemented to understand what the class offers.

Recreating the ScaleTo Animation

You can also animate the scale of your `AddWidgetFrame` control with the following:

```
public void ScaleTo()
{
    var animation = new Animation(v => AddWidgetFrame.
    Scale = v, 0, 1);
    animation.Commit(AddWidgetFrame, "ScaleTo");
}
```

When creating an instance of the `Animation` class, you provide the following parameter:

```
v => AddWidgetFrame.Scale = v
```

This is the `callback` parameter, and it allows you to define what property is set during the animation.

The next parameter is `start`. This is the starting value that will be passed into the callback lambda you defined in the first parameter. In your example, you set it to 0, meaning the `AddWidgetFrame` control will not be visible because it has a scale of 0.

The final parameter you pass in is `end`. This is the resulting value that will be passed into the callback lambda.

The animation will only begin when you call the `Commit` method. This method also allows you to define how long it should take as well as how often to call the callback parameter you defined.

```
animation.Commit(AddWidgetFrame, "ScaleTo", length: 2000);
```

This code shows the simplest type of animation you can create within .NET MAUI. It is entirely possible to create much more complex animations. To achieve this, you need to create an animation and then add child animations in order to define the changes for each property and different sequences in the animation.

Creating a Rubber Band Animation

As an example on how to build a complex animation, I would like to show you one of my favorite animations: the rubber band animation. This animation simulates the `VisualElement` being pulled horizontally, letting go, and then bouncing back to its original shape just like a rubber band would. Figure 9-3 shows what it would look like, albeit in motion.

***Figure 9-3.** The distinguishing frames from the animation you will be building*

Let's build the animation with the `Animation` class using the understanding you gained in the previous section.

```
public void Rubberband(VisualElement view)
{
    var animation = new Animation();
    animation.Add(0.00, 0.30, new Animation(v => view.
    ScaleX = v, 1.00, 1.25));
    animation.Add(0.00, 0.30, new Animation(v => view.
    ScaleY = v, 1.00, 0.75));
```

```
    animation.Add(0.30, 0.40, new Animation(v => view.
    ScaleX = v, 1.25, 0.75));
    animation.Add(0.30, 0.40, new Animation(v => view.
    ScaleY = v, 0.75, 1.25));
    animation.Add(0.40, 0.50, new Animation(v => view.
    ScaleX = v, 0.75, 1.15));
    animation.Add(0.40, 0.50, new Animation(v => view.
    ScaleY = v, 1.25, 0.85));
    animation.Add(0.50, 0.65, new Animation(v => view.
    ScaleX = v, 1.15, 0.95));
    animation.Add(0.50, 0.65, new Animation(v => view.
    ScaleY = v, 0.85, 1.05));
    animation.Add(0.65, 0.75, new Animation(v => view.
    ScaleX = v, 0.95, 1.05));
    animation.Add(0.65, 0.75, new Animation(v => view.
    ScaleY = v, 1.05, 0.95));
    animation.Add(0.75, 1.00, new Animation(v => view.
    ScaleX = v, 1.05, 1.00));
    animation.Add(0.75, 1.00, new Animation(v => view.
    ScaleY = v, 0.95, 1.00));
    animation.Commit(view, "RubberbandAnimation",
    length: 2000);
}
```

Yes, I know this looks quite different to the previous animation you built. Let's deconstruct the parts that feel unfamiliar.

```
animation.Add(0.00, 0.30, new Animation(v => view.ScaleX = v,
1.00, 1.25));
animation.Add(0.00, 0.30, new Animation(v => view.ScaleY = v,
1.00, 0.75));
```

The two lines above define the first transition in your animation. You see that the ScaleX property will change from 1.00 (100%) to 1.25 (125%) and the ScaleY property will change from 1.00 (100%) to 0.75% (75%) of the control's current size. This provides the appearance that the view is being stretched. The key new part for you is the use of the Add method and the first two parameters. This allows you to add the animation defined as the third parameter as a child of the animation it is being added to. The result is that when you Commit the main animation, all of the child animations will be executed based on the sequence you defined in these two first parameters. Let's cover what these parameters mean.

The first parameter is the beginAt parameter. This determines when the child animation being added will begin during the overall animation sequence. So in the example of your first line, you define 0.00, meaning it will begin as soon as the animation starts.

The second parameter is the finishAt parameter. This determines when the child being added will finish during the overall animation sequence. So in the example of your first line, you define 0.30, meaning it will end 30% into the animation sequence.

Both the beginAt and finishAt parameters should be supplied as a value between 0 and 1 and considered a percentage in the overall animation sequence. You will also notice that I tend to include the decimal places even when they are 0; this really makes it easier to read the animation sequence as it ensures that all of the code is indented in the same way.

Finally, you call the Commit method as before to begin the animation sequence.

Now that you have covered building animations and some possible examples of using them, let's combine them with your trigger knowledge to really make your AddWidgetFrame look great when it becomes visible.

Combining Triggers and Animations

Animations are a really powerful tool, but they require view knowledge. This is where having the ability to trigger them from a trigger allows you to keep with the MVVM approach and keep your view and view model cleanly separated.

Now that you have covered how to apply an animation to your overlay view and looked at separating view from view model through the use of triggers, you can combine the two together to trigger the animation and keep the separation.

Let's return to the ShowOverlayTriggerAction.cs file and add in the animation from the "Concurrent Animations" section (changes are in **bold**).

```
namespace WidgetBoard.Triggers;

public class ShowOverlayTriggerAction :
TriggerAction<VisualElement>
{
    public bool ShowOverlay { get; set; }
    protected override async void Invoke(VisualElement sender)
    {
        if (ShowOverlay)
        {
            sender.Scale = 0;
            sender.IsVisible = true;
            sender.Opacity = 0;
            await Task.WhenAll(
                sender.FadeTo(1),
                sender.ScaleTo(1, 500, Easing.SpringOut));
        }
```

```
        else
        {
            await sender.ScaleTo(0, 500, Easing.SpringIn);
            sender.Opacity = 0;
            sender.IsVisible = false;
        }
    }
}
```

The trigger action now provides two key visual changes when the ShowOverlay property value changes. When the property becomes true, the AddWidgetFrame control will both fade in over 250 milliseconds and scale up from 0 to 1 over 500 milliseconds. You also make use of the Easing.SpringOut option to give a slightly more fluid feel to the changes in the animation.

When ShowOverlay becomes false, you just reverse the scale animation to show it shrink. Once the animation has completed, you then make sure that the control is no longer visible.

This concludes the sections on triggers and animations. You have seen how they can help to both simplify the views and view models you create while at the same time provide some really great functionality to make your applications feel alive. I would recommend taking the application for a spin and observing the animations in action; sadly we can't show that functionality off in printed form.

Behaviors

Quite often as developers we need to extend functionality of controls; there are typically two approaches for this when you consider doing this to control:

1. Inheritance: You create a subclass of the control and add your functionality in there.
2. Composition: You create a class that can be used/reused by other classes in order to share functionality.

There are positives and negatives for both approaches and quite often reasons for choosing either. In Chapter 7, when we built our `BoardLayout` control we inherited from .NET MAUI controls, this was using inheritance to extend existing functionality. In this section, we will be using composition and creating a behavior that can be attached to controls.

In our application, we provide the ability for users to create their own boards by providing a name, number of columns, and number of rows. Currently the application enables/disables the Save button based on whether the user has entered a name; we will now take this further to provide some styling to highlight which field is invalid.

Creating Our Behavior

First, let's create a new folder in our project.

- Right-click the *WidgetBoard* project.
- Select **Add ➤ New Folder**.
- Click **Add**.

Next let's add a new class to the Behaviors folder and call it `RequiredStringValidationBehavior`.

In .NET MAUI, we attach behaviors to controls, and it is during the attaching that the behavior is responsible for initializing itself, and then the opposite is true when detaching – the behavior is responsible for tidying up to ensure the code does not lead to a memory leak. Let's incrementally add some code to our new class and see how the above can be applied.

First, we make our class inherit from the Behavior base class provided by .NET MAUI; see the changes in **bold**:

```
public class RequiredStringValidationBehavior : Behavior<Entry>
{
}
```

Next we can add two properties to our class; these will allow developers to define a valid and invalid Style.

```
    public Style? ValidStyle { get; set; }

    public Style? InvalidStyle { get; set; }
```

Then we can override the OnAttachedTo and OnDetachingFrom methods to initialize and tidy up, respectively. You will notice there are two overloads for each of the methods named above; one takes the Entry type and one takes BindableObject. We want to use the Entry ones because we want to access the TextChanged event on the Entry class.

```
    protected override void OnAttachedTo(Entry bindable)
    {
        base.OnAttachedTo(bindable);

        bindable.TextChanged += BindableOnTextChanged;
    }

    protected override void OnDetachingFrom(Entry bindable)
    {
        base.OnDetachingFrom(bindable);

        bindable.TextChanged -= BindableOnTextChanged;
    }
```

Finally, we can handle the TextChanged event and set the Style property on the Entry that this behavior is attached to, to either the InvalidStyle or the ValidStyle based on whether the user has entered any text.

```
    private void BindableOnTextChanged(object? sender,
    TextChangedEventArgs e)
    {
        if (sender is Entry entry &&
            InvalidStyle is not null &&
            ValidStyle is not null)
        {
            entry.Style = string.IsNullOrWhiteSpace(e.
            NewTextValue) ? InvalidStyle : ValidStyle;
        }
    }
```

Behaviors can be attached to multiple controls at once; therefore, it is recommended to consider this detail when designing your behavior.

Attaching Our Behavior

Now that we have created a behavior, we need to proceed to using it. Open up the *BoardDetailsPage.xaml* file and add the following code between the ContentPage and Border elements:

```
<ContentPage.Resources>
    <Style TargetType="Entry" x:Key="ValidEntryStyle">
        <Setter Property="BackgroundColor"
        Value="Transparent" />
    </Style>

    <Style TargetType="Entry" x:Key="InvalidEntryStyle">
        <Setter Property="BackgroundColor" Value="Red" />
    </Style>
</ContentPage.Resources>
```

We covered styles earlier; here we are adding a local `Style` which will only apply in the current page. Next we need to attach the behavior and assign the styles to it.

We can delete the following:

```
<Entry Text="{Binding BoardName}">
```

and replace it with

```
<Entry Text="{Binding BoardName}">
    <Entry.Behaviors>
        <behaviors:RequiredStringValidationBehavior
            ValidStyle="{StaticResource ValidEntryStyle}"
            InvalidStyle="{StaticResource
            InvalidEntryStyle}" />
    </Entry.Behaviors>
</Entry>
```

This creates a new instance of the `RequiredStringValidationBehavior` class and attaches it to the name `Entry`.

This concludes how to create a relatively simple behavior and attach it to a control in our application. For much more complex behaviors, I would thoroughly recommend making use of the .NET MAUI Community Toolkit, which provides a rich set of behaviors, including a more advanced version of the behavior we have created here.

Taking the Application for a Spin

We can now run the application and observe that when adding a new board, initially the name field will not show the invalid style; this is considered a nice approach for users so they aren't bombarded with a lot of red/warnings on the screen before they have started entering data. You usually only show them validation once they have started typing. Figure 9-4 shows the name entry with a red background after the user has deleted all text.

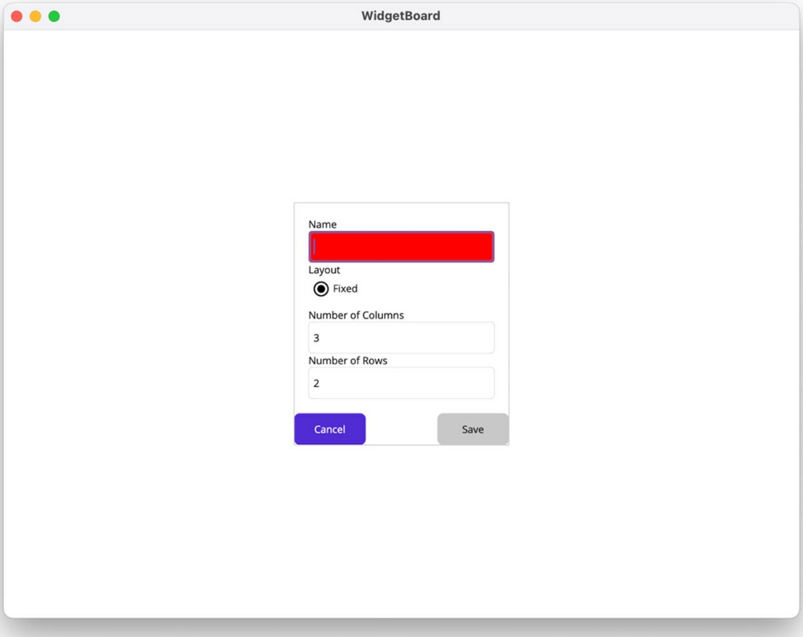

***Figure 9-4.** The name entry with a red background after the user has deleted all text*

The user can then enter text into the name field. Figure 9-5 shows the name entry with a white background after the user has entered some text.

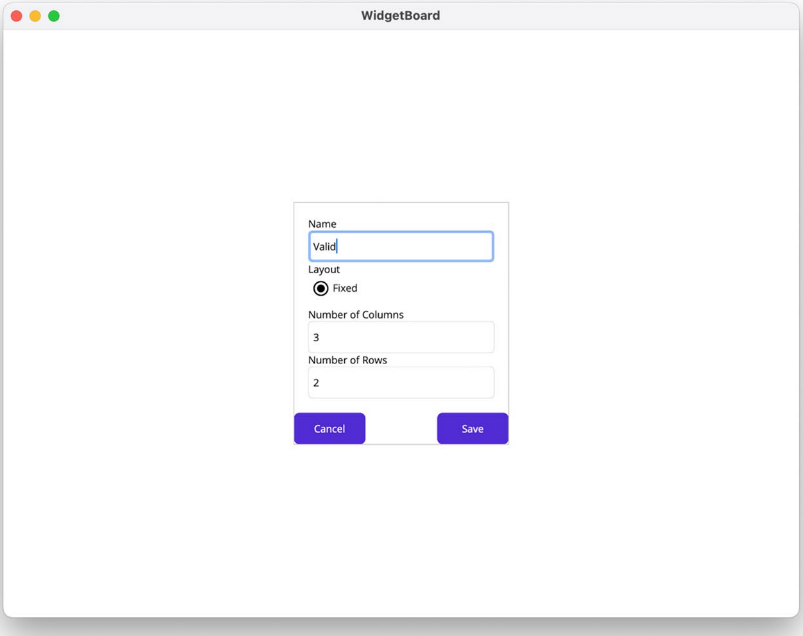

Figure 9-5. *The name entry with a red background after the user has deleted all text*

Fonts

We covered the topic of fonts in Chapter 3, but in this chapter, we are going to put it into action and apply a custom font to the clock widget. I always like the old digital display to show a clock. Sadly due to licensing issues, I haven't found a free font that exactly matches the digital display. The best I could find is the VT323 font. For the purpose of including it in your application, you can download it from `https://fonts.google.com/specimen/VT323` or feel free to choose any other font that you prefer. Once you have a font downloaded, let's proceed to using it in the application.

Embed the Font

The first step is to embed the font in the application; this is just a case for placing the *.ttf* or *.otf* file into the */Resources/Fonts/* folder.

Configure the Font

The next step is to configure it for use within .NET MAUI; you can do this by opening the *MauiProgram.cs* file and adding the following line into the `ConfigureFonts` method:

```
fonts.AddFont("VT323-Regular.ttf", "VT323");
```

This makes the font from the *VT323-Regular.ttf* file available for use under the alias of "VT323".

Use the Font

The next step is to make use of the new font in the *ClockWidgetView.xaml* file. Let's open that file and add the following line onto the `Label` element:

```
FontFamily="VT323"
```

The above matches the provided `FontFamily` name to the alias we provided when configuring the font. Let's see how this looks.

Taking the Application for a Spin

If we open the application one last time in this chapter, navigate to a board, and add the clock widget to the board, we will see that the clock widget now renders the new font. Figure 9-6 shows the clock widget rendering the new font.

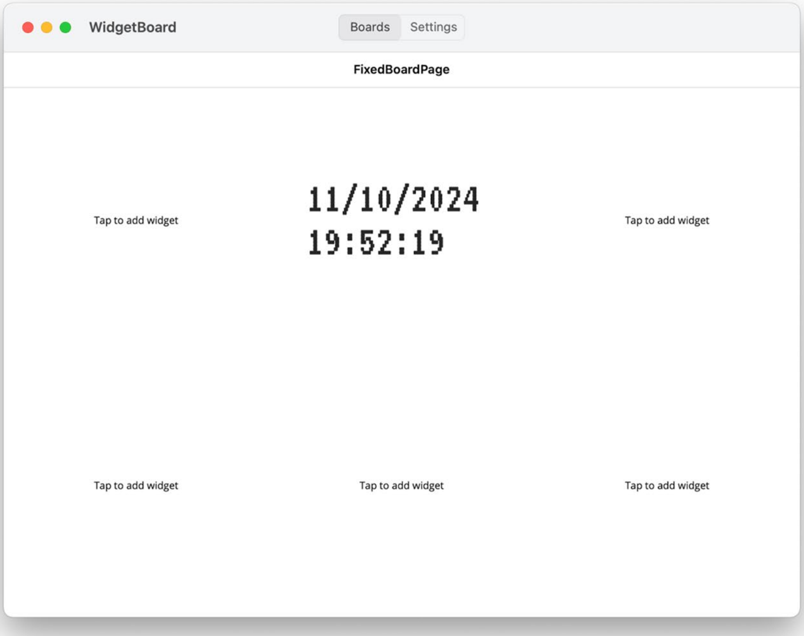

Figure 9-6. *The clock widget rendering the new VT323 font*

You should notice that the font chosen is a monospace font, which means all characters take up the same space; this is especially useful in this scenario because it will prevent the text from moving from side to side when the time changes.

Summary

In this chapter, you have

- Provided the ability to add a widget to a board
- Covered the different options available when showing an overlay

- Explored how you can define styling information for your application
- Learned how to handle devices running in light and dark modes
- Learned how to apply triggers to enhance your UI
- Covered how to animate parts of your application
- Explored what happens when you combine triggers and animations
- Created and attached a behavior
- Embedded a font

In the next chapter, you will

- Learn about the different types of local data
- Discover what .NET MAUI offers in terms of local file storage locations and when to use each one
- Gain an understanding of database technologies and apply two different options
- Modify your application to save and load the boards your users create
- Gain an understanding of the options for storing small bits of data or preferences
- Add the ability to record the last opened board
- Gain an understanding of the options for storing small bits of data securely or SecureStorage

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch09`.

Extra Assignment

I think you can take these animations to another level and really make your application feel alive! Try the following possible extensions!

Animate the BoxView Overlay

You've added an animation to present your `Border` with the widget selection details inside. A nice further enhancement on this would be to also animate the `BoxView` that you are using as your semi-transparent overlay. I personally think a nice `FadeTo` animation would work well, but I would love to hear what works best for you.

Animate the New Widget

To really make the application feel alive, you could consider animating each widget as it is added onto the board. You have the `Widgets_ChildAdded` method inside your `BoardLayout.xaml.cs` file where you set the `Position`. You could consider expanding this method implementation to also animate the new widget. Perhaps you could make the new widget scale up similar to how your `Border` presents.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch09-extra`.

CHAPTER 10

Local Data

Abstract

In this chapter, you will learn about the different types of local data, what they are best used for, and how to apply them in your application. The options will include understanding when and where to store data that needs to be kept secure.

You will modify your application to store the boards that your user creates so that they can be displayed in the slide-out menu and also be opened. You will also record the last opened board so that when returning to the application, this board will be presented to the user.

What Is Local Data?

When building an application, whether it is targeted at a single or at multiple platforms, you will very likely need to store data that represents the state of the application. The types of data you will need to store can vary between storing "simple" settings, caching files/data, and even storing a full set of data inside a local database. These types of data are called *local data* since they live on the device that your application is running on. Data that comes from a remote endpoint is called *remote data*, and this will be covered in Chapter 11.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_10

.NET MAUI provides multiple options when you want to store data locally on a device. Each option is better suited to a specific purpose and size of data. Here is a brief overview of those options:

- File system: Stores loose files directly on the device through file system access
- Database: Stores data in a file optimized for access
- Preferences: Stores data in key-value pairs
- Secure storage: Stores data in key-value pairs like preferences but stores them in a secure location on the device

File System

.NET MAUI provides some helpful abstractions over the multiple platforms that it supports. One such abstraction is the `FileSystem` helper class. It comes from the old Xamarin.Essentials library and now is a core part of .NET MAUI. It allows you to obtain useful bits of information to help with common tasks involving the file system.

Let's take a look at the properties the `FileSystem` class offers you as it helps to know when they should be used and for what type of data.

Cache Directory

You have no need to cache anything as part of the application we're building in this book; however, I feel this is a valuable piece of information to mention. This property enables you to get the most appropriate location to store cache data. You can store any type of data in this directory. Typically you store it when you want to persist it longer than just holding it in memory, but your application **must not** rely on this data to function because the operating system can and will purge the contents of this directory.

App Data Directory

The `AppDataDirectory` property provides the app's top-level directory for storing any files. These files are backed up with the operating system syncing framework (e.g., iCloud for Apple devices, Google Drive for Android, and OneDrive for Windows).

This property is precisely what you are going to use when creating and opening your database files in the next section. So let's set up the bits that you will need.

The `FileSystem` helper class provides a set of static properties, meaning you can simply write

```
var appDataDirectory = FileSystem.AppDataDirectory;
```

However, as you have discovered already in this book, it does not lend itself well to unit testing. Instead, you can rely on the `IFileSystem` interface and register the .NET MAUI implementation with your app builder. Let's open up your `MauiProgram.cs` file and add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton(FileSystem.Current);
```

This will register the `FileSystem.Current` property as the `IFileSystem` interface, so whenever you state that your classes depend on `IFileSystem`, they will be provided with the `FileSystem.Current` instance.

Now that you have covered `FileSystem` and are ready to create your database files, you can learn about database access in .NET MAUI.

Database

A database is a collection of data that is organized. In a database, data is organized or structured into tables consisting of rows and columns. Databases are a much better approach than storing data in files. The ability to index the data makes it easier to query and manipulate. There are different kinds of databases, ranging from relational databases to distributed databases, cloud databases, and NoSQL databases. In this chapter, you will focus on relational and NoSQL databases.

Every application I have ever built has required some form of database, and I suspect that most of the applications that you will build will also require one. In fact, a customer once insisted that we build them an application without a database until we helped them understand the true value that a database provides. A database really provides value when you need to link data together or filter and sort the data in an efficient manner.

In your application, you are going to provide the ability to save a board and return a list of boards that the user has created. You will also provide the ability to store where the widgets have been placed so that they will be remembered when a user loads the board back up. This means that we will need to store information about a widget and the board that they belong to. There are many ways to structure this, and if you are not familiar with database design approaches and how to optimize the design, I would strongly recommend reading up on the subject along with database normalization. Figure 10-1 shows the entity relationship diagram for the database you will be creating.

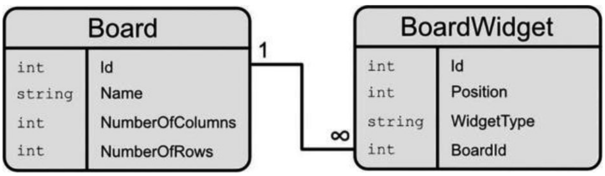

***Figure 10-1.** The entity relationship diagram of your database models*

To abstract this approach slightly, you will be using the repository pattern.

Repository Pattern

The repository pattern allows you to hide all the logic that deals with creating, reading, updating, and deleting (also known as CRUD) entities within your application. By using this pattern, it allows you to keep all the knowledge around how entities are loaded, saved, and more in a single place. This has the added benefit that if you want to completely change where your data is loaded from, you only need to change the implementation inside the repository. It also allows you to provide mock implementations when wanting to perform things like unit testing and you don't want to have to rely on an actual database existing. The repository pattern will also work well in this application due to the number of different parts of the application that perform the same or similar things (e.g., loading a list of boards). Having a single repository that does all of the database-related activities makes it easier to maintain as the application grows in size and complexity.

Let's add a new folder called `Data` and then add an interface for your repository to that folder called `IBoardRepository`. Change the code to look as follows:

```
using WidgetBoard.Models;

namespace WidgetBoard.Data;

public interface IBoardRepository
{
    void CreateBoard(Board board);

    void CreateBoardWidget(BoardWidget boardWidget);

    void DeleteBoard(Board board);
```

```
    IReadOnlyList<Board> ListBoards();

    Board? LoadBoard(int boardId);

    void UpdateBoard(Board board);
}
```

Now that you have defined your interface, you can update your application's code base to use this interface when loading and saving your boards.

Creating a Board

Thankfully our application only provides one location to create a board; I would like to argue that makes for a good design practice because it keeps the creation logic in a single place.

The first place you will update is your `BoardDetailsPageViewModel` class, which provides support for creating a new board. Open up the class and make the following modifications.

Add a new `IBoardRepository` field.

```
private readonly IBoardRepository boardRepository;
```

Assign a valid instance to the `boardRepository` field; the modifications are in **bold**.

```
public BoardDetailsPageViewModel(
    ISemanticScreenReader semanticScreenReader,
    IBoardRepository boardRepository)
{
    this.semanticScreenReader = semanticScreenReader;
    this.boardRepository = boardRepository;
    SaveCommand = new Command(
        () => Save(),
        () => !string.IsNullOrWhiteSpace(BoardName));
}
```

Use the boardRepository field when saving; the modifications are in **bold**.

```
private async void Save()
{
    var board = new Board
    {
        Name = BoardName,
        NumberOfColumns = NumberOfColumns,
        NumberOfRows = NumberOfRows
    };

    this.boardRepository.CreateBoard(board);
    semanticScreenReader.Announce($"A new board with the name
    {BoardName} was created successfully.");

    await Shell.Current.GoToAsync(
        RouteNames.FixedBoard,
        new Dictionary<string, object>
        {
            { "Board", board }
        });
}
```

That concludes the changes required to provide support for creating a board in the application. Let's move onto loading the list of boards from the repository.

Listing Your Boards

In the previous chapters, you just added a fixed list of boards and added them to the Boards collection in your `AppShellViewModel`, `BoardListPageViewModel`, and `BoardSearchHandler` classes. Now you are going to modify those classes so they can be populated by the boards the user creates and you store in the database. Let's do each one in turn because they each have a slightly different approach needed to update them to follow a good practice.

Load the List in AppShellViewModel

Open the `AppShellViewModel.cs` file and make the following changes.

Add a field for your `IBoardRepository.`

```
private readonly IBoardRepository boardRepository;
```

Modify your constructor to use the `IBoardRepository` as a dependency.

```
public AppShellViewModel(
    IBoardRepository boardRepository)
{
    this.boardRepository = boardRepository;
}
```

Load the list of boards and populate your collection.

```
public void LoadBoards()
{
    Boards.Clear();

    var boards = this.boardRepository.ListBoards();
    foreach (var board in boards)
    {
        Boards.Add(board);
    }
}
```

There is a further change that you need to make in order to allow your `AppShellViewModel` class to actually load the board. You need to hook into some of the lifecycle events that apply to Pages in .NET MAUI. `AppShell` inherits from Page, which means you get full access to those lifecycle events. The specific event you care about now is the OnAppearing event. It is called when your page is displayed on screen.

The OnAppearing method can be called multiple times during the lifetime of the page, so it is recommended to make your method idempotent or check whether it has been called before in order to prevent odd behavior when called a second time.

OnAppearing is a great choice for your scenario because it will result in your code being executed every time the view appears; this can be every time your flyout menu is opened. This provides you with the ability to refresh your list of boards every time the user opens the flyout menu. The main reason it is fine for your scenario is because you will be loading data from a local database with a limited number of boards to load, so it will be pretty quick. In scenarios where you are loading from an external web service, it can take much more time to perform it, and therefore, you may wish to maintain some level of caching and prevent calling the web service every time the view appears. A better option under this scenario and probably most typical scenarios in .NET MAUI applications is to use the OnNavigatedTo method.

Let's open your AppShell.xaml.cs file and make use of this lifecycle method.

```
protected override void OnAppearing()
{
    base.OnAppearing();
    ((AppShellViewModel)BindingContext).LoadBoards();
}
```

When the method gets called, you use the newly added LoadBoards method on your view model. The main reason you hook into this lifecycle event is when you eventually try to navigate to the last used board in the LoadBoards method, you need to make sure the application has started rendering; otherwise, the navigation will fail.

Load the List in BoardListPageViewModel

These changes will look very familiar to the previous section; we are repeating them to make sure nothing gets missed, and then you will make use of a different approach when calling the `LoadBoards` method.

Open the `BoardListPageViewModel.cs` file and make the following changes.

Add a field for your `IBoardRepository`.

```
private readonly IBoardRepository boardRepository;
```

Modify your constructor to use the `IBoardRepository` as a dependency.

```
public BoardListPageViewModel (
    IBoardRepository boardRepository)
{
    this.boardRepository = boardRepository;
}
Load the list of boards and populate your collection.
public void LoadBoards()
{
    Boards.Clear();

    var boards = this.boardRepository.ListBoards();
    foreach (var board in boards)
    {
        Boards.Add(board);
    }
}
```

As we learned in the previous section, it is typically much better practice to use the `OnNavigatedTo` method rather than the `OnAppearing` method to load data. This is due to the fact that `OnAppearing` can be called multiple times during the lifetime of a page, which can introduce unexpected behavior. Therefore, you will be making use of the `OnNavigatedTo` method in the *BoardListPage.xaml.cs* file; let's open the file and make the following changes:

```
protected override void OnNavigatedTo(NavigatedToEvent
Args args)
{
    base.OnNavigatedTo(args);
    ((BoardListPageViewModel)BindingContext).LoadBoards();
}
```

Load the List in BoardSearchHandler

This is starting to feel a little repetitive right? I agree; don't worry, we aren't going to repeat the same steps from before, because we can actually reuse the exact code from the previous section! Thankfully because the `BoardSearchHandler` class is a descendant of the `BindableObject` property, we can add a `BindableProperty` onto the class just like we did in Chapter 6 and bind to the `Boards` property we are already populating in the `BoardListPageViewModel` class. Let's apply these in a step-by-step process now.

Open the *BoardSearchHandler.cs* file and make the following changes:

– Delete the `boards` field that was being created and initialized with three hard-coded boards.

– Add the following `BindableProperty`:

```
public static readonly BindableProperty
BoardsProperty =
    BindableProperty.Create(
        nameof(Boards),
        typeof(ObservableCollection<Board>),
        typeof(BoardSearchHandler));
```

```
public ObservableCollection<Board> Boards
{
    get => (ObservableCollection<Board>)
    GetValue(BoardsProperty);
    set => SetValue(BoardsProperty, value);
}
```

– Change the `OnQueryChanged` method implementation to use the new `Boards` property (change in **bold**), and yes it is just a case of changing a single character.

```
protected override void OnQueryChanged(string oldValue,
string newValue)
{
    base.OnQueryChanged(oldValue, newValue);
    if (string.IsNullOrWhiteSpace(newValue))
    {
        ItemsSource = null;
    }
    else
    {
        ItemsSource = Boards
            .Where(board => board.Name.
            Contains(newValue, StringComparison.
            CurrentCultureIgnoreCase))
```

```
                    .ToList<Board>();
            }
        }
```

That was a different set of changes to the previous loading of data up, but quite often you will find there are handy ways to share data that has already been loaded into your application. This makes it possible to build apps that will perform better and consume less memory due to the nature of sharing what has already been loaded.

The final change is to open the BoardListPage.xaml file and make the following modification (in **bold**):

```
<Shell.SearchHandler>
    <widgetBoard:BoardSearchHandler
        Boards="{Binding Boards}"
        Placeholder="Enter board name"
        ShowsResults="True"
        DisplayMemberName="Name" />
</Shell.SearchHandler>
```

This will now populate our new `Boards` property on the `BoardSearchHandler` class with the `Boards` value from the `BoardListPageViewModel` class.

Loading a Board

Up until this point you have relied on passing the `Board` into the `FixedBoardPageViewModel` and displaying the details of that. The loading process would become rather inefficient if you were to load all boards and the associated `BoardWidgets` when listing all boards in the system, so you need to do this in a two-step process: first, list the boards as you did in the previous section and, second, load the board in the view model. This will be a slightly involved process, so let's walk through it step by step. Open the `FixedBoardPageViewModel.cs` file and make the following changes.

Add the following fields to store the board that is loaded and the repository to perform the load:

```
private Board? board;
private readonly IBoardRepository boardRepository;
```

In your constructor, add the board repository dependency and assign to the newly created field. Changes are in **bold**.

```
public FixedBoardPageViewModel(
    WidgetTemplateSelector widgetTemplateSelector,
    WidgetFactory widgetFactory,
    IBoardRepository boardRepository)
{
    WidgetTemplateSelector = widgetTemplateSelector;
    this.widgetFactory = widgetFactory;
    this.boardRepository = boardRepository;

    Widgets = new ObservableCollection<IWidgetViewModel>();

    AddWidgetCommand = new Command(OnAddWidget);
    AddNewWidgetCommand = new Command<int>(index =>
    {
        IsAddingWidget = true;
        addingPosition = index;
    });
}
```

Now let's load the Board inside your ApplyQueryAttributes method. The changes are in **bold**.

```
public void ApplyQueryAttributes(IDictionary<string,
object> query)
{
    var boardParameter = query["Board"] as Board;
    board = boardRepository.LoadBoard(boardParameter.Id);

    if (board is not null)
    {
        BoardName = board.Name;
        NumberOfColumns = board.NumberOfColumns;
        NumberOfRows = board.NumberOfRows;

        foreach (var boardWidget in board.BoardWidgets)
        {
            var widgetViewModel = widgetFactory.CreateWidget
            ViewModel(boardWidget.WidgetType);

            if (widgetViewModel is null)
            {
                continue;
            }

            widgetViewModel.Position = boardWidget.Position;
            Widgets.Add(widgetViewModel);
        }
    }
}
```

Next, add the ability to save a widget's position on the board.

```
private void SaveWidget(IWidgetViewModel widgetViewModel)
{
    if (board is null)
    {
        return;
    }

    var boardWidget = new BoardWidget
    {
        BoardId = board.Id,
        Position = widgetViewModel.Position,
        WidgetType = widgetViewModel.Type
    };
    boardRepository.CreateBoardWidget(boardWidget);
}
```

The above method will create a new BoardWidget model class and save it into the database for you.

Finally, you need to call the SaveWidget method. For the purpose of your application, you are going to provide an autosave feature, so each time a widget is added to the board, you will save it immediately to the database. In order to achieve this, you just need to add the **bold** line into your AddWidget method.

```
private void OnAddWidget()
{
    if (SelectedWidget is null)
    {
        return;
    }

    var widgetViewModel = widgetFactory.CreateWidgetViewModel(
    SelectedWidget);
```

```
    if (widgetViewModel is not null)
    {
        widgetViewModel.Position = addingPosition;
        Widgets.Add(widgetViewModel);

        SaveWidget(widgetViewModel);
    }

    IsAddingWidget = false;
}
```

You can't run your code yet because you don't have an implementation of your `IBoardRepository` interface, so let's look at two different database options that will allow you to provide an implementation for your `IBoardRepository`.

SQLite

SQLite is a lightweight cross-platform database that has become the go-to option for providing database support in mobile applications. The database is stored locally in a single file on the device's file system.

SQLite is supported natively by Android and iOS; however, they require access via C++. There are several C# wrappers around the native SQLite engine that .NET developers can use. The most popular choice is the C# wrapper called SQLite-net.

Installing SQLite-net

In order to install and use SQLite-net, you need to install the NuGet package called *Sqlite-net-pcl*. You may notice the extra *-pcl* suffix in the NuGet package name and find this confusing. This is an artifact of an old piece of technology used in Xamarin.Forms applications. The name has been retained, but don't worry; this is the correct package for adding to a .NET MAUI project.

You can do this by following these steps:

1. Right-click the WidgetBoard project.
2. Click **Manage NuGet Packages**.
3. In the Search field, enter *Sqlite-net-pcl*.
4. Select the **Sqlite-net-pcl package** and select **Add Package**.
5. A confirmation dialog will show. Review and accept the license details if you are happy.

Using SQLite-net

The first step is to create your IBoardRepository implementation. Add a new class file called SqliteBoardRepository in your Data folder, and make it implement your IBoardRepository interface.

```
using SQLite;
using WidgetBoard.Models;

namespace WidgetBoard.Data;

public class SqliteBoardRepository : IBoardRepository
{
    public void CreateBoard(Board board)
    {
        throw new NotImplementedException();
    }
    public void CreateBoardWidget(BoardWidget boardWidget)
    {
        throw new NotImplementedException();
    }
```

```
    public void DeleteBoard(Board board)
    {
        throw new NotImplementedException();
    }
    public IReadOnlyList<Board> ListBoards()
    {
        throw new NotImplementedException();
    }
    public Board? LoadBoard(int boardId)
    {
        throw new NotImplementedException();
    }
    public void UpdateBoard(Board board)
    {
        throw new NotImplementedException();
    }
}
```

You also need to register your implementation with the app builder in `MauiProgram.cs`. You can add the following line into the `CreateMauiApp` method:

```
builder.Services.AddTransient<IBoardRepository,
SqliteBoardRepository>();
```

Connecting to an SQLite Database

As mentioned, an SQLite database is contained within a single file, so when connecting to the database, you need to provide the path to that file. You can do this through the `SqliteConnection` class. Note that if you wish to make use of async/await, you can use the `SqliteAsyncConnection` class.

Let's edit your repository class to support opening a connection to your database.

Add a field for the database connection.

```
private readonly SQLiteConnection connection;
```

Add a constructor to open the connection.

```
public SqliteBoardRepository(IFileSystem fileSystem)
{
    var dbPath = Path.Combine(fileSystem.AppDataDirectory,
    "widgetboard_sqlite.db");
    connection = new SQLiteConnection(dbPath);
}
```

Here you make use of the `IFileSystem` implementation you registered in the previous section. Then you make use of it to determine where to store your database file. Finally, you open a connection using the path to your database file. Note that if the file does not exist, one will be created for you.

Mapping Your Models

The SQLite-net library provides the ability to define mapping information in your model classes that will ultimately be used to create your table definition automatically for you. There is a rich set of options ranging from setting a `PrimaryKey` through to defining if a column has a `MaxLength` or even if it needs to be `Unique`. Open your `Board.cs` file and make the following modifications in **bold**:

```
using SQLite;

namespace WidgetBoard.Models;

public class Board
{
    [PrimaryKey, AutoIncrement]
```

```
    public int Id { get; set; }
    public string Name { get; init; }
    public int NumberOfColumns { get; init; }
    public int NumberOfRows { get; init; }
    [Ignore]
    public IReadOnlyList<BoardWidget> BoardWidgets { get;
    set; } = [];
}
```

You add a new ID column, marking it as the PrimaryKey, and state that it will AutoIncrement, meaning that SQLite-net will manage the ID generation for you. You have also added the BoardWidgets property and marked it with the Ignore attribute; this tells the SQLite-net library not to map this property onto the database table – we will handle this property ourselves.

Your second model class is in the BoardWidget.cs file. This represents each widget that is placed on the board and where it is positioned.

```
using SQLite;

namespace WidgetBoard.Models;

public class BoardWidget
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public int BoardId { get; set; }
    public int Position { get; set; }
    public string WidgetType { get; set; } = string.Empty;
}
```

Creating Your Tables

You can inform the SQLite-net connection to create a table for you. This can be done by calling the `CreateTable<T>` method and passing the appropriate model type. Note that `CreateTable` is idempotent, so unless you change your model, calling `CreateTable` a second time will have no impact. You can modify your `SqliteBoardRepository` to call the `CreateTable` method in its constructor as follows (changes in **bold**):

```
public SqliteBoardRepository(IFileSystem fileSystem)
{
    var dbPath = Path.Combine(fileSystem.AppDataDirectory,
    "widgetboard_sqlite.db");
    connection = new SQLiteConnection(dbPath);
    connection.CreateTable<Board>();
    connection.CreateTable<BoardWidget>();
}
```

Inserting into an SQLite Database

You can now add in the ability to insert a board into your database by supplying the following implementation into the `CreateBoard` method:

```
public void CreateBoard(Board board)
{
    connection.Insert(board);
}
```

The same approach can be applied to the `CreateBoardWidget` method:

```
public void CreateBoardWidget(BoardWidget boardWidget)
{
    connection.Insert(boardWidget);
}
```

Reading a Collection from an SQLite Database

You only need to return a list of the boards your user has created in the application.

```
public IReadOnlyList<Board> ListBoards()
{
    return connection.Table<Board>()
        .ToList();
}
```

Perhaps you should consider sorting these boards alphabetically. SQLite-net offers a rich set of functionality when querying data in the database. You can make use of LINQ-based expressions, which gives you the following (the addition in **bold**):

```
public IReadOnlyList<Board> ListBoards()
{
    return connection.Table<Board>()
        .OrderBy(b => b.Name)
        .ToList();
}
```

Note that sorting the list in this way will be more efficient than loading the list into memory and then sorting the items.

Reading a Single Entity from an SQLite Database

When reading a `Board` from the database, you also need to load any `BoardWidgets` that relate to it. For this, you can write the following:

```
public Board? LoadBoard(int boardId)
{
    var board = connection.Find<Board>(boardId);
```

```
    if (board is null)
    {
        return null;
    }

    var widgets = connection.Table<BoardWidget>().Where(w =>
    w.BoardId == boardId).ToList();

    board.BoardWidgets = widgets;
    return board;
}
```

The first line calling `Find` allows you to find an entity with the supplied primary key value. This retrieves the `Board`. Next, you need to retrieve the collection of `BoardWidgets`. This is performed in a very similar manner to loading your collection of `Boards`. Finally, you assign the widgets you loaded into the board before returning it to the caller.

It is worth noting that the *Sqlite-net-pcl* package does not provide more complex querying operations such as joins. If this is something that you still require, it is possible to write the SQL directly and execute against the connection. If you wish to join your `Board` and `BoardWidget` tables together, you can achieve this as follows:

```
var board = connection.Query<Board>("SELECT B.* FROM Board B
JOIN BoardWidget BW ON BW.BoardId = B.BoardId WHERE B.BoardId =
?", boardId);
```

Note that the above query is purely aimed at showing how joins work; it does not provide you with any particularly useful in the context of your application.

Deleting from an SQLite Database

While I haven't focused on providing this functionality just yet, it is a very common use case.

```
public void DeleteBoard(Board board)
{
    connection.Delete(board);
}
```

Updating an Entity in an SQLite Database

While I haven't focused on providing this functionality just yet, it is a very common use case.

```
public void UpdateBoard(Board board)
{
    connection.Update(board);
}
```

This concludes the section on adding an SQLite-based database into our .NET MAUI application. You or your team may decide that SQLite is not the approach that you wish to take; therefore, the next section will provide you with an alternative approach.

LiteDB

LiteDB is a simple, fast, and lightweight embedded .NET document database. LiteDB was inspired by the MongoDB database, and its API is very similar to the official MongoDB .NET API.

Installing LiteDB

In order to install and use LiteDB, you need to install the NuGet package called *LiteDB*. Don't worry; it is perfectly fine to install both the LiteDB and SQLite packages side by side into your project. In fact, that is precisely what you will do here.

You can do this by following these steps:

1. Right-click the WidgetBoard project.
2. Click **Manage NuGet Packages**.
3. In the Search field, enter *LiteDB*.
4. Select the **LiteDB package** and select **Add Package**.
5. A confirmation dialog will show. Review and accept the license details if you are happy.

Using LiteDB

The first step is to create your `IBoardRepository` implementation. Add a new class file called `LiteDBBoardRepository` in your `Data` folder, and make it implement your `IBoardRepository` interface.

```
using LiteDB;
using WidgetBoard.Models;

namespace WidgetBoard.Data;

public class LiteDBBoardRepository : IBoardRepository
{
    public void CreateBoard(Board board)
    {
        throw new NotImplementedException();
    }
```

```
    public void CreateBoardWidget(BoardWidget boardWidget)
    {
        throw new NotImplementedException();
    }
    public void DeleteBoard(Board board)
    {
        throw new NotImplementedException();
    }
    public IReadOnlyList<Board> ListBoards()
    {
        throw new NotImplementedException();
    }
    public Board? LoadBoard(int boardId)
    {
        throw new NotImplementedException();
    }
    public void UpdateBoard(Board board)
    {
        throw new NotImplementedException();
    }
}
```

You also need to register your implementation with the app builder in `MauiProgram.cs`. You can add the following line. Just make sure that you have removed or commented out the line to register the `SqliteBoardRepository` implementation.

```
builder.Services.AddTransient<IBoardRepository,
LiteDBBoardRepository>();
```

Connecting to a LiteDB Database

LiteDB stores all its data in a single file on disk, so your first task is to specify where this file exists so that you can create and open the file for users within your application. For this part, you will borrow a concept from a little further ahead in this chapter (the "File System" section).

Edit your repository class to support opening a connection to your database.

Add a field to hold the database access details.

```
private readonly LiteDatabase database;
```

Add a constructor to open the connection.

```
public LiteDBBoardRepository(IFileSystem fileSystem)
{
    var dbPath = Path.Combine(fileSystem.AppDataDirectory,
    "widgetboard_litedb.db");
    database = new LiteDatabase(dbPath);
}
```

The above should look very similar to the SQLite way of accessing the database. Here you make use of the `IFileSystem` implementation you registered in the previous section. Then you make use of that to determine where to store your database file. Finally, you open a connection using the path to your database file. Note that if the file does not exist, one will be created for you.

Mapping Your Models

First, you need to add a field to hold a collection of boards and one for the collection of board widgets.

```
private readonly ILiteCollection<Board> boardCollection;
private readonly ILiteCollection<BoardWidget>
boardWidgetCollection;
```

Then you need to get access to that collection in order to allow you to perform your operations against it.

```
boardCollection = database.GetCollection<Board>("Boards");
boardWidgetCollection
= database.GetCollection<BoardWidget>("BoardWidgets");
```

The final part of your mapping setup is to define indexing information about your model. For this, you use the `EnsureIndex` method.

```
boardCollection.EnsureIndex(b => b.Id, true);
```

In LiteDB, any property that you wish to be unique or want to query against needs to have a definition provided through the `EnsureIndex` method. This should feel like a familiar concept to any relational database developers that are used to creating keys or indexes on their database tables.

Creating Your Tables

You don't actually need to do anything to create your tables here. The key difference between LiteDB and other databases that you might use is that the schema of the data is held with the data.

Inserting into a LiteDB Database

You can now add in the ability to insert a board into your database by supplying the following implementation into the `CreateBoard` method:

```
public void CreateBoard(Board board)
{
    boardCollection.Insert(board);
}
```

The same approach can be applied to the CreateBoardWidget method:

```
public void CreateBoardWidget(BoardWidget boardWidget)
{
    boardWidgetCollection.Insert(boardWidget);
}
```

Reading a Collection from a LiteDB Database

You only need to return a list of the boards your user created in the application.

```
public IReadOnlyList<Board> ListBoards()
{
    return boardCollection.Query()
        .ToList();
}
```

Perhaps you should consider sorting these boards alphabetically. LiteDB offers a similar set of functionality that you looked at with SQLite-net. LINQ-based expressions can be used to order your boards, which gives you the following (the addition is in **bold**):

```
public IReadOnlyList<Board> ListBoards()
{
    return boardCollection.Query()
        .OrderBy(b => b.Name)
        .ToList();
}
```

You also need to add the following line to your constructor to make sure querying is possible:

```
boardCollection.EnsureIndex(b => b.Name, false);
```

Reading a Single Entity from a LiteDB Database

When reading a `Board` from the database, you also need to load any `BoardWidgets` that relate to it. For this, you can write the following:

```
public Board? LoadBoard(int boardId)
{
    var board = boardCollection.FindById(boardId);

    if (board is null)
    {
        return null;
    }

    var boardWidgets = boardWidgetCollection.Find(w =>
    w.BoardId == boardId).ToList();
    board.BoardWidgets = boardWidgets;

    return board;
}
```

The first line calls the `FindById` method, which allows you to find an entity with the supplied primary key value. This retrieves the `Board`. Next, you need to retrieve the collection of `BoardWidgets`. This is performed in a very similar manner to loading your collection of `Boards`. Finally, you assign the widgets you loaded into the board before returning it to the caller.

Deleting from a LiteDB Database

While I haven't focused on providing this functionality, it is a very common use case.

```
public void DeleteBoard(Board board)
{
    boardCollection.Delete(board.Id);
}
```

Updating an Entity in a LiteDB Database

While I haven't focused on providing this functionality, it is a very common use case.

```
public void UpdateBoard(Board board)
{
    boardCollection.Update(board);
}
```

This concludes the section on adding a LiteDB-based database into our .NET MAUI application.

Database Summary

There is an abundance of options when it comes to choosing not only which database but then also the Object Relational Mapping (ORM) layer on top of it. The aim of this section is to give a taste of what some options offer and to encourage you to decide which will benefit your application and team most.

Both options I covered provide support for encryption; SQLite requires that you install an additional package called *sqlite-net-sqlcipher*, and LiteDB supports encryption out of the box.

I strongly encourage you to evaluate which database will provide you with the best development experience and the users of your application with the best user experience. Some databases perform better in different scenarios.

Moving forward with this application, you will continue to use LiteDB.

Application Settings (Preferences)

Quite often you will want to persist data about your application that you really do not need a database for. I like to refer to these bits of data as application settings. If you have previous experience with building .NET applications, this would be similar to an `app.config` or `appsettings.json` file. The .NET MAUI term is *Preferences*, though, and this is the API that you will look at accessing.

An item in Preferences is stored as a key-value pair. The key is a string, and it is recommended to keep the name short in length.

As with all of the other APIs provided by .NET MAUI, you will register the Preferences implementation with the app builder in the `MauiProgram.cs` file. You can add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton(Preferences.Default);
```

Now let's proceed to looking at what types of data can be stored in *Preferences*.

What Can Be Stored in Preferences?

There is a limitation on the type of data that can be stored in Preferences. The API provides the ability to store the following .NET types:

- `Boolean`
- `Double`
- `Int32`
- `Single`
- `Int64`
- `String`
- `DateTime`

Having the ability to provide a `String` value surely means you could in theory store anything in there, right? While this is technically possible, it is highly recommended that you only store small amounts of text. Otherwise, the performance of storing and retrieval can be impacted in your applications.

Setting a Value in Preferences

You can store a value in Preferences through the use of the `Set` method. You can provide a key, the value, and also an optional `sharedName`. The preferences stored in your application are only visible to that application. You can also create a shared preference that can be used by other extensions or a watch application should you wish.

A perfect use case for your application is to store the ID of the last accessed board and open it the next time the application loads. Let's store the ID initially. Inside your `FixedBoardPageViewModel` class, you can make the following changes.

Add the preferences field.

```
private readonly IPreferences preferences;
```

Update the constructor to set the preferences field (changes in **bold**).

```
public FixedBoardPageViewModel(
    WidgetTemplateSelector widgetTemplateSelector,
    WidgetFactory widgetFactory,
    IBoardRepository boardRepository,
    IPreferences preferences)
{
    WidgetTemplateSelector = widgetTemplateSelector;
    this.preferences = preferences;
    Widgets = new ObservableCollection<IWidgetViewModel>();
}
```

Finally, record the ID of the board that was supplied when navigating to the page. You can do this by adding the **bold** line to your `ApplyQueryAttributes` method:

```
public void ApplyQueryAttributes(IDictionary<string,
object> query)
{
    var boardParameter = (Board)query["Board"];
    board = boardRepository.LoadBoard(boardParameter.Id);

    if (board is not null)
{
        preferences.Set("LastUsedBoardId", board.Id);
        BoardName = board.Name;
        NumberOfColumns = board.NumberOfColumns;
        NumberOfRows = board.NumberOfRows;
    }
}
```

This means that every time a user opens a board to view it, the ID will be remembered in Preferences. When the application is opened again in the future, it will use that ID to open the last viewed board.

A possible alternative way of achieving this type of functionality could be to maintain a last opened column in the database and always find the latest of that set.

Getting a Value in Preferences

You can retrieve a value from Preferences using the `Get` method. You are required to supply the key identifying the setting and a default value to be returned if the key does not exist. You can optionally provide a `sharedName`, much like with the Set method covered in the previous section.

You have already written the code to store your `LastUsedBoardId` in Preferences, so let's read it back when loading your boards up to display. Open up your `AppShellViewModel.cs` file and make the following changes.

Add the following fields:

```
private readonly IPreferences preferences;
private readonly IDispatcher dispatcher;
```

Set the preferences field in the constructor (changes in **bold**).

```
public AppShellViewModel(
    IBoardRepository boardRepository,
    IPreferences preferences,
    IDispatcher dispatcher)
{
    this.boardRepository = boardRepository;
    this.preferences = preferences;
    this.dispatcher = dispatcher;
}
```

Update your `LoadBoards` method to support navigating to the last used board (changes in **bold**).

```
public void LoadBoards()
{
    Boards.Clear();

    var boards = this.boardRepository.ListBoards();
    var lastUsedBoardId = preferences.
    Get("LastUsedBoardId", -1);
    Board? lastUsedBoard = null;
    foreach (var board in boards)
    {
        Boards.Add(board);
        if (lastUsedBoardId == board.Id)
```

```
        {
            lastUsedBoard = board;
        }
    }
    if (lastUsedBoard is not null)
    {
        dispatcher.Dispatch(() =>
        {
            BoardSelected(lastUsedBoard);
        });
    }
}
```

There are a few new concepts here, so let's break them down into understandable chunks.

First is the use of the `preferences.Get` method, as you learned about before writing the above code. You supply the key name and the default value to be returned if the key does not exist. You use -1 for the default because it is not a valid ID for a database key.

The final new concept is the use of the `IDispatcher` implementation provided by .NET MAUI. This allows you to trigger a deferred action and make sure that it is dispatched onto the UI thread. Your method will be called on the UI thread, but you want the `OnAppearing` logic to finish before you attempt to navigate somewhere; by calling `dispatcher.Dispatch`, you are queuing up an action to be performed once the UI thread is no longer busy. .NET MAUI does handle a lot of dispatching for you when you trigger updates in bindings, but there are times when you need to make sure that you are updating things on the UI thread.

If you run your code now, you can create a new board and view it once saved. If you then close and reopen the application, you will see that the board you created is now shown for you. Providing an experience like this can go a long way to an enjoyable user experience (UX) as they are returning to where they were previously.

Checking If a Key Exists in Preferences

There can be times when you are unable to supply a suitable default value to the Get method in order to know whether a value has been set, for example, using a Boolean. false is a valid value, and therefore, the default value would not be able to distinguish whether it was set as false or the default value of false. In this scenario, you can make use of the ContainsKey method. So instead of writing

```
var lastUsedBoardId = preferences.Get("LastUsedBoardId", -1);
```

you could have first checked whether the key existed, like

```
if (preferences.ContainsKey("LastUsedBoardId"))
{
    // Perform your logic
}
```

Removing a Preference

There may be times when you need to remove an option from the Preferences store or even remove all options. If you want to remove your LastUsedBoardId preference, you can write

```
Preferences.Remove("LastUsedBoardId");
```

If you want to remove all options, you can write

```
Preferences.Clear();
```

Displaying Our Preferences

Now is the time to finally add some content into the SettingsPage class. We can add in the ability to display the current value for the LastUsedBoardId and allow the user to clear it. Let's open the *SettingsPage.xaml* file and add the following changes (in **bold**):

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Pages.SettingsPage"
    x:DataType="viewModels:SettingsPageViewModel">
    <VerticalStackLayout Padding="20">
        <Label
            Text="Last used board"
            VerticalOptions="Center" />

        <HorizontalStackLayout>
            <Label
                Text="{Binding LastUsedBoard}"
          MinimumWidthRequest="200"
                VerticalOptions="Center" />
            <Button
                Text="Clear"
                Command="{Binding ClearLastUsedBoardCommand}"
                SemanticProperties.Hint="Clears the last used
                board value from settings. This means the
                application won't automatically load a board
                when opened." />
        </HorizontalStackLayout>
    </VerticalStackLayout>
</ContentPage>
```

The above makes use of all the good practices that we have covered so far: adding in compiled bindings, including accessibility information through `SemanticProperties`, etc. The key detail is that we have added a `Label` which will display the name of the last used board and a `Button` to allow for clearing this value.

Next we need to assign the SettingsPageViewModel to the BindingContext of the page. Let's open the *SettingsPage.xaml.cs* file and do that (changes in **bold**).

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Pages;

public partial class SettingsPage : ContentPage
{
    public SettingsPage(SettingsPageViewModel viewModel)
    {
        InitializeComponent();
        BindingContext = viewModel;
    }
}
```

The final change is to modify the *SettingsPageViewModel.cs* file to provide the properties and logic to assign values to those properties. The resulting file should look as follows (with changes in **bold**):

```
using System.Windows.Input;
using WidgetBoard.Data;

namespace WidgetBoard.ViewModels;

public class SettingsPageViewModel : BaseViewModel
{
    public SettingsPageViewModel(
        IPreferences preferences,
        IBoardRepository boardRepository)
    {
        var lastUsedBoardId = preferences.
        Get("LastUsedBoardId", -1);
```

```
        if (lastUsedBoardId != -1)
        {
            LastUsedBoard = boardRepository.
            LoadBoard(lastUsedBoardId)?.Name ?? string.Empty;
        }

        ClearLastUsedBoardCommand = new Command(() =>
        {
            preferences.Remove("LastUsedBoardId");
            LastUsedBoard = string.Empty;
        });
    }

    private string lastUsedBoard = string.Empty;

    public string LastUsedBoard
    {
        get => lastUsedBoard;
        set => SetProperty(ref lastUsedBoard, value);
    }

    public ICommand ClearLastUsedBoardCommand { get; }
}
```

By now, most of these changes should start to feel familiar; we are

- Adding a property (`LastUsedBoard`) that will notify the UI of any changes
- Adding a command that the UI can bind to in order to execute an action in the view model
- Getting the "`LastUsedBoardId`" value from `Preferences`
- Using that value to load a specific board and assign its name to the `LastUsedBoard` property

This concludes how you can store, load, and remove application settings. Now let's proceed to learning about how to secure application settings.

Secure Storage

When building an application, there will quite often be an occasion where you need to store an API token or some form of data that needs to be held securely. .NET MAUI provides another API that makes sure that the values you supply are held securely on each of the platforms' secure storage locations.

As always with a new API provided by .NET MAUI, you must register it with the `MauiAppBuilder` in your `MauiProgram.cs` file, so let's open up that file and add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton(SecureStorage.Default);
```

This will allow you to declare a dependency on `ISecureStorage` in your class constructors and have it provided for you. Next you can add in the functionality to the `SettingsPage` just as you did for the `Preferences` implementation.

You don't currently have a need to use a secure value just yet. It will follow in the next chapter, but given that we are covering the storage of local data, we can make the changes to the application ready for using it in Chapter 11. Let's open up the `SettingsPage.xaml` file and add in the following section below the existing `</HorizontalStackLayout>` element:

```
<Label
    Text="Open Weather API token"
    VerticalOptions="Center" />

<HorizontalStackLayout>
    <Entry
```

```
        Text="{Binding OpenWeatherApiToken}"
        MinimumWidthRequest="200"
        IsPassword="True" />
        <Button
        Text="Save"
        Command="{Binding SaveApiTokenCommand}"
        SemanticProperties.Hint="Saves the currently entered
        Open Weather API token into secure storage." />
</HorizontalStackLayout>
```

As before, most of this should feel familiar; one key detail to highlight is the use of the `IsPassword` property on the `Entry` element – this allows you to add an entry field that will mask the entered characters with the * character and therefore protect the value from prying eyes. Now that we have added in the UI, let's open the *SettingsPageViewModel.cs* file and actually interact with the `ISecureStorage` API.

Add the following fields and properties to the class:

```
private string openWeatherApiToken = string.Empty;

public string OpenWeatherApiToken
{
    get => openWeatherApiToken;
    set => SetProperty(ref openWeatherApiToken, value);
}

public ICommand SaveApiTokenCommand { get; }
```

And then you can modify the constructor to look as follows (with changes in **bold**):

```
public SettingsPageViewModel(
    IPreferences preferences,
    IBoardRepository boardRepository,
```

```
    ISecureStorage secureStorage)
{
    var lastUsedBoardId = preferences.
    Get("LastUsedBoardId", -1);
    if (lastUsedBoardId != -1)
    {
        LastUsedBoard = boardRepository.
        LoadBoard(lastUsedBoardId)?.Name ?? string.Empty;
    }

    ClearLastUsedBoardCommand = new Command(() =>
    {
        preferences.Remove("LastUsedBoardId");
        LastUsedBoard = string.Empty;
    });

    SaveApiTokenCommand = new Command(async () =>
    {
        await secureStorage.SetAsync("OpenWeatherApiToken",
        OpenWeatherApiToken);
    });

    OpenWeatherApiToken = secureStorage.GetAsync(
    "OpenWeatherApiToken").GetAwaiter().GetResult() ??
    string.Empty;
}
```

To summarize, you are providing the following.

Storing a Value Securely

The application will save a value in secure storage with the key of `OpenWeatherApiToken` and the value entered by the user when the user clicks the Save button. This is covered by the following line from the above changes:

```
await secureStorage.SetAsync("OpenWeatherApiToken",
OpenWeatherApiToken);
```

Reading a Secure Value

The application will also load the currently stored value against the key of OpenWeatherApiToken and display the value inside the entry field.

This is covered by the following line from the previous changes:

```
OpenWeatherApiToken = secureStorage.GetAsync(
"OpenWeatherApiToken").GetAwaiter().GetResult() ??
string.Empty;
```

Removing a Secure Value

As with Preferences, you can remove all secure values.

To remove a specific value, remove the key:

```
bool success = SecureStorage.Default.
Remove("OpenWeatherApiToken ");
```

To remove all values, use the RemoveAll method:

```
SecureStorage.Default.RemoveAll();
```

Platform Specifics

As mentioned, the SecureStorage API makes use of each of the platform-specific APIs to handle the actual storage of the data you pass in. It is worth noting that the implementations for each individual platform are different and may change in the operating systems but SecureStorage will leverage whatever is in the operating system and therefore will always be the most secure option. This section explains how.

Android

The data you pass in is encrypted with the Android `EncryptedSharedPreferences` class, from the Android Security library, which automatically encrypts keys and values using a two-scheme approach:

1. Keys are deterministically encrypted so that the key can be encrypted and properly looked up.
2. Values are nondeterministically encrypted using AES-256 GCM.

The Android Security library provides an implementation of the security best practices related to reading and writing data at rest, as well as key creation and verification.

Since Google introduced Android 6.0 (API level 23), the operating system offers the ability to back up the user's data. This includes the `Preferences` and also the `SecureStorage` that .NET MAUI offers. It is entirely possible, and in fact, I recommend that you disable this backup functionality when using `SecureStorage`.

In order to disable the auto backup feature, you need to set the `android:allowBackup` to `false` in the `AndroidManifest.xml` file under the `Platforms/Android` folder. The resulting change should look something like the following:

```
<manifest ... >
    ...
    <application android:allowBackup="false" ... >
        ...
    </application>
</manifest>
```

iOS and macOS

Data passed into SecureStorage on iOS and macOS is encrypted through the Keychain API. To quote Apple:

> *The keychain is the best place to store small secrets, like passwords and cryptographic keys. You use the functions of the keychain services API to add, retrieve, delete, or modify keychain items.*

For further reading, refer to the Apple documentation at https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_keychain.

In some cases, keychain data is synchronized with iCloud, and uninstalling the application may not remove the secure values from user devices. I have certainly observed this in some applications I have built, so it is best to plan around this possibility.

Windows

SecureStorage on Windows uses the DataProtectionProvider class to encrypt values securely. The .NET MAUI implementation allows for the data to be protected against the local user or computer account.

For further reading, refer to the Microsoft documentation at https://docs.microsoft.com/uwp/api/windows.security.cryptography.dataprotection.dataprotectionprovider?view=winrt-22621.

Viewing the Result

Now when running your application, you will see that not only does the last board that you create get loaded back up but it also shows the widgets you previously added. Figure 10-2 shows an example of the results.

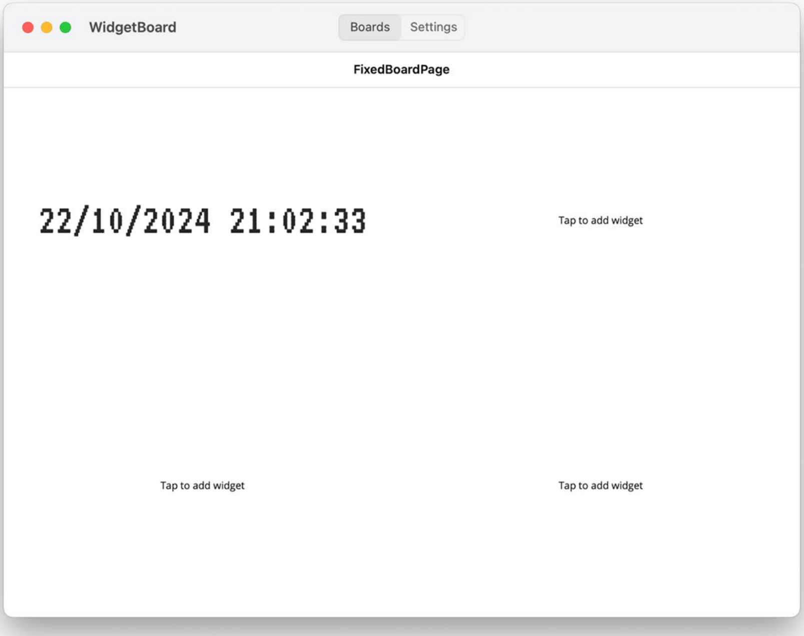

Figure 10-2. *The application loads back up and shows the previously added widgets*

Then selecting the Settings tab will present the user with the new settings-based page you just added. Figure 10-3 shows an example of the settings tab page.

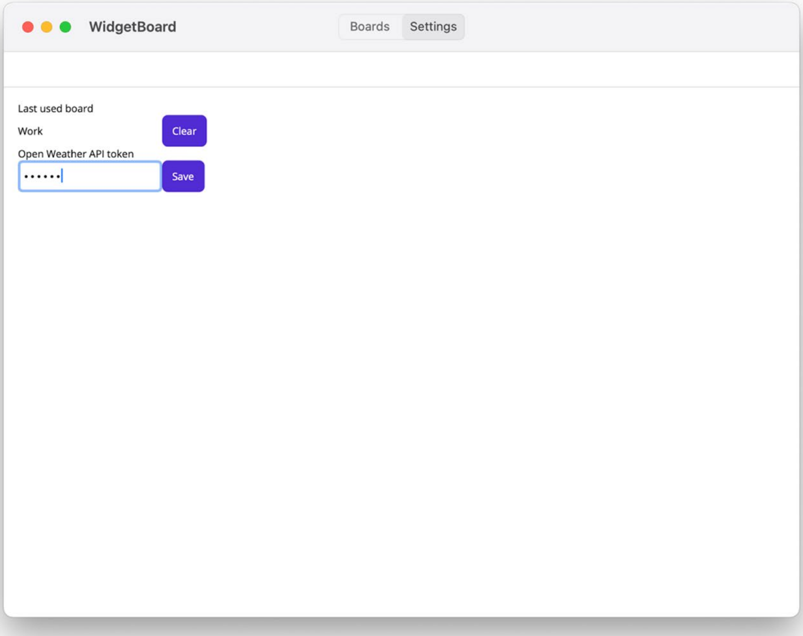

Figure 10-3. *The application showing the settings page to the user*

Summary

In this chapter, you have

- Learned about the different types of local data
- Discovered what .NET MAUI offers in terms of local file storage locations and when to use each one
- Gained an understanding of database technologies and applied two different options
- Modified your application to save and load the boards your users create

- Gained an understanding of the options for storing small bits of data or Preferences
- Added the ability to record the last opened board
- Gained an understanding of the options for storing small bits of data securely or SecureStorage

In the next chapter, you will

- Learn about remote data
- Learn how you can interact with it
- Cover the common considerations
- Look at a concrete example with the Open Weather API
- Build your own implementation to consume the Open Weather API
- Cover how to consume the data returned
- Talk through scenarios where things can go wrong
- Provide implementations to handle those scenarios
- Look at how you can reduce the complexity of your implementation with Refit
- Add in your Weather Widget

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch10`.

Extra Assignment

You have provided the ability for users to add widgets to their boards and automatically save them so when they next load the board, it will be remembered for them. I would like to see if you can add the ability to remove the widgets from the board and the database.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch10-extra`.

CHAPTER 11

Remote Data

Abstract

In this chapter, you will be exploring the topic of remote data, learning what exactly it is, types of it, how to interact with it, and what to consider when doing so. You will then build upon this learning by building a new widget, the Weather Widget, to display the current weather. This will be done by interacting directly with the Open Weather API. You will get exposure to handling HTTP requests and responses with an API, how to handle the response being in a JSON format, and the varying levels of flexibility when mapping to the JSON data. You will finish off by simplifying the implementation with a fantastic NuGet package that generates source code for you, simply from an interface you define to represent the web service.

What Is Remote Data?

Remote data is any data that is sourced from outside of your application; typically it isn't even on the device where the application is installed and running. This can range from querying a web API in order to obtain data, utilizing a cloud-based database provider, images hosted online, streaming video or audio data, and more.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_11

The vast majority of applications will interact with some form of remote endpoint in order to pull data. In this world of constantly changing data, this becomes an essential part of practically any application.

Considerations When Handling Remote Data

There can be quite a few concepts to consider when interacting with remote data. You will be explicitly addressing these as you build your new widget, but I want to draw your attention to them before you start.

Loading Times

One of the worst experiences for a user is to tap on a button or open a new page/application and just see the application lock up while it is loading data. The user will think that the application has crashed, and in fact, platforms like Android and Windows will likely indicate that the application has crashed/locked up if the load takes too long. Thankfully .NET offers you the `async` and `await` keywords. They are not essential, but they really do make your life easier. There could be an entire chapter or even book on this topic; however, my good friend Brandon Minnick has already covered a lot of this in his AsyncAwaitBestPractices repository on GitHub. We will be covering the basics in order to build a responsive application. I thoroughly recommend you do if you want to dig deeper (`https://github.com/brminnick/AsyncAwaitBestPractices`).

A common use case is to display a visual that makes it clear to the user that the application is busy loading. This can be with a simple `ActivityIndicator`, which loads the platform-specific spinner/loading icon users should feel familiar with, or you can make use of the animation features I covered in Chapter 9 to show something more involved. With this loading display, you then initiate your web service call. If you get a response, you display the result of that response in your application (e.g., items in a shopping list or, in your scenario, the user's current weather).

Failures

During the building of a recent application, some of the most valuable testing a friend provided for me was to install the application and then ride the London Underground and observe just how flaky a mobile phone's data connection really can be.

There are two key questions to consider when dealing with network connectivity issues:

1. What does the user need to know?
2. How does the application need to recover?

Don't worry, I will be covering examples of how to answer these questions as we build our weather widget.

Security

As a developer of applications, it is essential that you maintain the trust that your users put in you with regard to keeping their data safe. With this in mind, you should always choose HTTPS over HTTP. In fact, most platforms won't allow HTTP traffic by default to avoid it accidentally being used. There are ways to disable the prevention of HTTP traffic; however, I strongly advise against it, so I won't cover how to do so in this book.

I strongly recommend that as you build your applications, you consider security as a top priority. The Open Web Application Security Project (OWASP) is a nonprofit foundation that works to improve the security of software, and it provides some really great resources and guidance on what you should consider when building websites and mobile applications. As a good starting point, look at their Mobile Application Security Testing Guide repository on GitHub at `https://github.com/OWASP/owasp-mastg/`.

Quite often APIs will require levels of authentication that complicate the flow to pulling data from them. This typically happens when your application needs to consume data specific to a user and not just the API itself. I won't be covering this scenario in this book, but I recommend reading up on OAuth 2.0 with a good initial resource at `www.oauth.com/oauth2-servers/mobile-and-native-apps/`. Additionally, specific APIs such as the GitHub API will likely provide good documentation on how to use their specific authentication mechanism. So with this in mind, I recommend referring to the documentation for the API that you wish to integrate with.

Web Services

Web services act as a mechanism to query or obtain data from a remote server. They typically offer many advantages to developers building them because they provide the ability to charge based on usage, protect the developer's intellectual property, and other reasons.

The Open Weather API

You will be calling the Open Weather API and specifically version 2.5 of the One Call API. The API is free to use with some usage limits. You can call it up to 60 times per minute and 1,000,000 calls per month, which will certainly be fine for this scenario.

For the initial work, you will be using a fixed latitude and longitude of `20.7984` and `-156.3319`, respectively, which, if you look it up, represents Maui, Hawaii. You will enable the application to use the device's current location information in the next chapter.

Creating an Open Weather Account

You will be required to create an account. To do so, navigate to the website at `https://home.openweathermap.org/users/sign_up` and create the account. Note that you do not need to enter any billing details. You can use it entirely for free. If you breach the call limits, the API will simply fail instead of running into accidental charges.

Creating an Open Weather API Key

Next, you need to create an API key, which can be done on the following page at https://home.openweathermap.org/api_keys. Keep a copy of this API key ready for when you eventually use it later in this chapter. Don't worry too much for now as you can return to the above web page and access the key.

Examining the Data

Before you dive into writing some code, you should take a look at the API and the data that it returns. In fact, the API offers a lot more detail than you really need. You can consume the details in case you want to use them in the future; however, this does bring in some possible drawbacks. It increases the complexity of reading through the data if you need to debug things, and it also increases the amount of data that needs to be retrieved by your application. In the mobile world, this can be expensive!

Given the above, you can make the following web service call which includes following details:

- Calls version 2.5 of the One Call API
- Supplies a latitude of `20.7984`
- Supplies a longitude of `-156.3319`
- Supplies units of `metric`, meaning you will receive degrees Celsius (apologies if you still work in imperial units)
- Supplies the API key you created in the previous section

The full URL that you need to call looks as follows:

https://api.openweathermap.org/data/2.5/weather?lat=20.7984&lon=-156.3319&units=metric&appid=APIKEY

You can open this in any web browser to view the following response back; just make sure to replace the APIKEY text with your own API key. You can see the key details that you will need for your application highlighted in **bold**.


```
{
    "coord": {
        "lon":-156.3319,
        "lat":20.7984
    },
    "weather":[
        {
            "id":802,
            "main":"Clouds",
            "description":"scattered clouds",
            "icon":"03d"
        }
    ],
    "base":"stations",
    "main": {
        "temp":22.73,
        "feels_like":22.96,
        "temp_min":21.23,
        "temp_max":24.1,
        "pressure":1017,
        "humidity":73,
        "sea_level":1017,
        "grnd_level":945
    },
    "visibility":10000,
    "wind": {
        "speed":3.09,
        "deg":300
    },
```

```
    "clouds": {
        "all":40
    },
    "dt":1729711746,
    "sys": {
        "type":2,
        "id":18862,
        "country":"US",
        "sunrise":1729700629,
        "sunset":1729742100
    },
    "timezone":-36000,
    "id":5852697,
    "name":"Pukalani",
    "cod":200
}
```


Using System.Text.Json

In order to consume and deserialize the contents of the JSON returned to you, you need to use one of the following two options:

- Newtonsoft.Json (requires a NuGet package)
- System.Text.Json

Newtonsoft has been around for many years and is a go-to option for many developers. System.Text.Json has become its successor and is my recommendation for this scenario, especially as it is backed by Microsoft and James Newton-King, the author of Newtonsoft, who works for Microsoft.

Let's go ahead and use *System.Text.Json* as it is the recommended way to proceed and is included with .NET MAUI out of the box.

Now that you have seen what the data looks like, you can start to build the model classes that will allow you to deserialize the response coming back from the API.

Creating Your Models

I highlighted that you really don't need all of the information that is returned from the API. Thankfully you only need to build your model to cover the detail that you require and allow the rest to be ignored during the deserialization process.

Let's create the model classes you require. You do this in the reverse order that they appear in the JSON due to the fact that the outer elements need to refer to the inner elements.

First, add a new folder to keep everything organized and call it `Communications`.

Now, add a new class file and call it `Weather.cs`.

```
namespace WidgetBoard.Communications;

public class Weather
{
    public string Main { get; set; } = string.Empty;
    public string Icon { get; set; } = string.Empty;
    public string IconUrl => $"https://openweathermap.org/img/
    wn/{Icon}@2x.png";
}
```

Your `Weather` class maps to the `weather` element in the JSON returned from the API. You can see that you are mapping to the `main` and `icon` elements and you have added a calculated property that returns a URL pointing to the icon provided by the Open Weather API. The last property you are mapping, `IconUrl`, is yet another great example of remote data. The API provides you with an icon that can be rendered inside your application representing the current weather of the location. Based on the example in your original JSON, you see the `icon` value of `03d`. This represents clouds and is making me feel a little happy knowing that a typically sunny Maui is cloudy when it is sunny here in the UK when it would typically be cloudy.

You will notice that the casing of your property names does not match the element names in the JSON. This will actually result in the deserialization process failing to map as you require. When you get to the deserialization part, you will see how to handle this scenario.

Your next model class to add should be called `Main,` and similarly to the `Weather` class, it will map to the element that matches its name: `main`. Your `Main` class file should have the following contents:

```
using System.Text.Json.Serialization;

namespace WidgetBoard.Communications;

public class Main
{
    [JsonPropertyName("temp")]
    public double Temperature { get; set; }
}
```

This class will currently only map to the current `Temperature`, there are many other values that you could map to if you wish to show more detail in your widget. With the `Temperature` property mapping, you can see how it is possible to map from a property in your model to an element in JSON that has a different name. This functionality is extremely valuable when building your own models because it allows you to name the properties to provide better context. I personally prefer to avoid abbreviations and stick with explicit names to make the intentions of the code clear.

Your final model class to add should be called `Forecast.cs` and will have the following contents:

```
namespace WidgetBoard.Communications;

public class Forecast
{
    public Main? Main { get; set; }
    public Weather[] Weather { get; set; } = [];
}
```

This class maps to the top-level element in the returned JSON. You are mapping to the `Timezone` element, the `Current`, which will contain your previously mapped values, and an array of `Weather` elements.

Now that you have created the model classes that can be mapped to the JSON returned from the Open Weather API, you can proceed to calling the API in order to retrieve that JSON.

Connecting to the Open Weather API

Before you start to build the implementation for accessing the API, you are going to create an interface to define what it should do. This has the added benefit that when you wish to unit test any class that depends on the `IWeatherForecastService`, you can supply a mock implementation rather than requiring that the unit tests will access the real API. I will cover why that is a bad idea in Chapter 14, but the simple answer here is that you have a limited number of calls you are allowed to make for free and you don't want unit tests eating that allowance up.

```
namespace WidgetBoard.Communications;

public interface IWeatherForecastService
{
    Task<Forecast?> GetForecast(double latitude, double
    longitude, string apiKey);
}
```

A common naming approach to classes that interact with APIs is to add the suffix *Service* to show that it provides a service to the user. Therefore, let's create your service by adding a new class file and calling it `WeatherForecastService.cs`. Add the following contents:

```
using System.Text.Json;

namespace WidgetBoard.Communications;

public class WeatherForecastService : IWeatherForecastService
{
    private readonly HttpClient httpClient;
    private const string ServerUrl = "https://api.
    openweathermap.org/data/2.5/weather?";

    public WeatherForecastService(HttpClient httpClient)
    {
        this.httpClient = httpClient;
    }

    public async Task<Forecast?> GetForecast(double latitude,
    double longitude, string apiKey)
    {
        var response = await httpClient
            .GetAsync($"{ServerUrl}lat={latitude}&lon=
            {longitude}&units=metric&appid={apiKey}")
            .ConfigureAwait(false);

        response.EnsureSuccessStatusCode();

        var stringContent = await response.Content
            .ReadAsStringAsync()
            .ConfigureAwait(false);
```

```
        var options = new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        };
        return JsonSerializer.Deserialize<Forecast>
        (stringContent, options);
    }
}
```

You added a fair amount into this class file, so let's walk through it step by step and cover what it does.

First is the `HttpClient` backing field, which is set within the constructor and will be supplied by the dependency injection layer. You also have a constant representing the URL of the API.

Next is the main piece of functionality in the `GetForecast` method. The first line in this method handles connecting to the Open Weather API and passing your latitude, longitude, and API key values. You also make sure to set `ConfigureAwait(false)` because you do not need to be called back on the initial calling thread. This helps to boost performance a little as it avoids having to wait until the calling thread becomes free.

```
var response = await httpClient.GetAsync($"{ServerUrl}lat=
{latitude}&lon={longitude}&units=metric&appid={apiKey}")
    .ConfigureAwait(false);
```

Then you make sure that the request was handled successfully by calling

```
response.EnsureSuccessStatusCode();
```

Note that the above will throw an exception if the status code received was not a 200 (success ok).

Then you extract the string content from the response.

```
var stringContent = await response.Content
    .ReadAsStringAsync()
    .ConfigureAwait(false);
```

Finally, you make use of the *System.Text.Json* library in order to deserialize the string content into the model classes that you created.

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true
};
return JsonSerializer.Deserialize<Forecast>(stringContent,
options);
```

I mentioned earlier that you had to explicitly opt-in to matching your property names to the JSON elements case-insensitively. You can see from the above code that you can do this through the use of the `JsonSerializerOptions` class and specifically the `PropertyNameCaseInsensitive` property.

Now that you have created the service, you should add your weather widget and make use of the service.

Creating the WeatherWidgetView

In order to create your widget, you need to add a new view. Add a new `.NET MAUI ContentView (XAML)` into your `Views` folder and call it `WeatherWidgetView`. This results in two files being created: `WeatherWidgetView.xaml` and `WeatherWidgetView.xaml.cs`. You need to update both files.

WeatherWidgetView.xaml

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

```
xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:viewModels="clr-namespace:WidgetBoard.ViewModels"
x:Class="WidgetBoard.Views.WeatherWidgetView"
x:DataType="viewModels:WeatherWidgetViewModel">
<VerticalStackLayout>
    <Label
        Text="Today"
        FontSize="20"
        VerticalOptions="Center"
        HorizontalOptions="Start"
        TextTransform="Uppercase" />
    <Label
        VerticalOptions="Center"
        HorizontalOptions="Center">
        <Label.FormattedText>
            <FormattedString>
                <Span
                    Text="{Binding Temperature,
                    StringFormat='{0:F1}'}"
                    FontSize="60"/>
                <Span
                    Text="°C" />
            </FormattedString>
        </Label.FormattedText>
    </Label>
    <Label
        Text="{Binding Weather}"
        FontSize="20"
        VerticalOptions="Center"
        HorizontalOptions="Center" />
```

```
        <Image
            Source="{Binding IconUrl}"
            WidthRequest="100"
            HeightRequest="100"/>
    </VerticalStackLayout>
</ContentView>
```

Some of the above XAML should feel familiar based on the previous code you have written. Some bits are new, so let's cover them.

`Label.FormattedText` enables you to define text of varying formats inside a single `Label` control. This can be helpful especially when parts of the text change dynamically in length and therefore result in the contents moving around. In your example, you are adding a `Span` with a text binding to your `Temperature` property in the view model and a second `Span` with the degrees Celsius symbol.

The second new concept is the use of `Image`. The binding on the `Source` property looks relatively straightforward; however, it is worth noting that .NET MAUI works some magic for you here. You are binding a string to the property. Under the hood, .NET MAUI converts the string into something that can resemble an image source. In fact, the underlying type is called `ImageSource`. Further to this, it will inspect your string, and if it contains a valid URL (e.g., starts with https://), then it will aim to load it as a remote image rather than looking in the application's set of compiled resources. .NET MAUI will also potentially handle caching of images for you to help reduce the amount of requests sent in order to load images from a remote source. In order to make use of this functionality, you need to provide a `UriImageSource` property on your view model rather than the `string` property.

The process of converting from one type to another is referred to as TypeConverters and can be fairly common in .NET MAUI. I won't go into detail on how they work, so please go to the Microsoft documentation site at https://learn.microsoft.com/dotnet/api/system.componentmodel.typeconverter.

WeatherWidgetView.xaml.cs

You also need to make the following adjustments to the `WeatherWidgetView.xaml.cs` file. This part is required because you haven't created a common base class for the widget views. At times there can be good reasons to create them; however, because you want to keep the visual tree as simple as possible, there isn't a common visual base class to use.

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public partial class WeatherWidgetView : ContentView,
IWidgetView
{
    public WeatherWidgetView()
    {
        InitializeComponent();
    }
    public IWidgetViewModel WidgetViewModel
    {
        get => (IWidgetViewModel)BindingContext;
        set => BindingContext = value;
    }
}
```

Now that you have created your widget view, you should create the view model that will be paired with it.

Creating the WeatherWidgetViewModel

The view model that you need to create in order to represent the weather-related data that can be bound to the UI requires some work that you are familiar with and some that you are not as familiar with. Let's proceed to adding the familiar bits and then walk through the newer concepts. First, add a new class file in the `ViewModels` folder and call it `WeatherWidgetViewModel.cs`. The initial contents should be modified to look as follows:

```
using WidgetBoard.Communications;

namespace WidgetBoard.ViewModels;

public class WeatherWidgetViewModel : BaseViewModel,
IWidgetViewModel
{
    public const string DisplayName = "Weather";
    public int Position { get; set; }
    public string Type => DisplayName;
}
```

The above should look familiar as it is very similar to the `ClockWidgetViewModel` you created earlier on in the book. Now you need to add in the weather-specific bits.

First, add a dependency on the `IWeatherForecastService` you created a short while ago and also the `ISecureStorage` implementation that we covered in the previous chapter.

```
private readonly IWeatherForecastService weatherForecastService;
private readonly ISecureStorage secureStorage;

public WeatherWidgetViewModel(IWeatherForecastService
weatherForecastService, ISecureStorage secureStorage)
{
    this.weatherForecastService = weatherForecastService;
    this.secureStorage = secureStorage;
    Task.Run(async () => await LoadWeatherForecast());
}
```

```
private async Task LoadWeatherForecast()
{
    var apiKey = await this.secureStorage.GetAsync("OpenWeather
    ApiToken");

    if (apiKey is null)
    {
        return;
    }

    var forecast = await weatherForecastService.
    GetForecast(20.798363, -156.331924, apiKey);

    if (forecast?.Main is null)
    {
        return;
    }

    Temperature = forecast.Main.Temperature;
    Weather = forecast.Weather.First().Main;
    IconUrl = forecast.Weather.First().IconUrl;
}
```

Inside of your constructor, you keep a copy of the service and you also start a background task to fetch the forecast information. Quite often you wouldn't start something like this from within a constructor; however, given that you know your view model will only be created when it is being added to the UI, this is perfectly acceptable.

Finally, you need to add the properties that your view wants to bind to.

```
private string iconUrl = string.Empty;
private double temperature;
private string weather = string.Empty;
```

```
public string IconUrl
{
    get => iconUrl;
    set => SetProperty(ref iconUrl, value);
}
public double Temperature
{
    get => temperature;
    set => SetProperty(ref temperature, value);
}
public string Weather
{
    get => weather;
    set => SetProperty(ref weather, value);
}
```

That's all you need in the view model for now. You can now register the widget and get it ready for your first test run.

Registering Your Widget

You first need to make use of a NuGet package in order to follow some recommended practices for the registration and usage of the `HttpClient` class. Go ahead and add the *Microsoft.Extensions.Http* NuGet package and then take a look at how to use it.

- Right-click the *WidgetBoard* solution.
- Select **Manage NuGet Packages**.
- Search for *Microsoft.Extensions.Http*.
- Select the correct package.
- Click **Add Package**.

Inside your `MauiProgram.cs` file, you need to add the following lines into the `CreateMauiApp` method:

```
builder.Services.AddHttpClient<WeatherForecastService>();
builder.Services.AddSingleton<IWeatherForecastService,
WeatherForecastService>();
WidgetFactory.RegisterWidget<WeatherWidgetView, WeatherWidgetVi
ewModel>(WeatherWidgetViewModel.DisplayName);
builder.Services.AddTransient<WeatherWidgetView>();
builder.Services.AddTransient<WeatherWidgetViewModel>();
```

The above code registers your widget's view and view models with the dependency injection layer and also registers it with your `WidgetFactory`, meaning it can be created from your add widget overlay.

Testing Your Widget

If you run your application, you will first need to navigate to the Settings tab and enter your Open Weather API key. You can see the API key obfuscated in Figure 11-1.

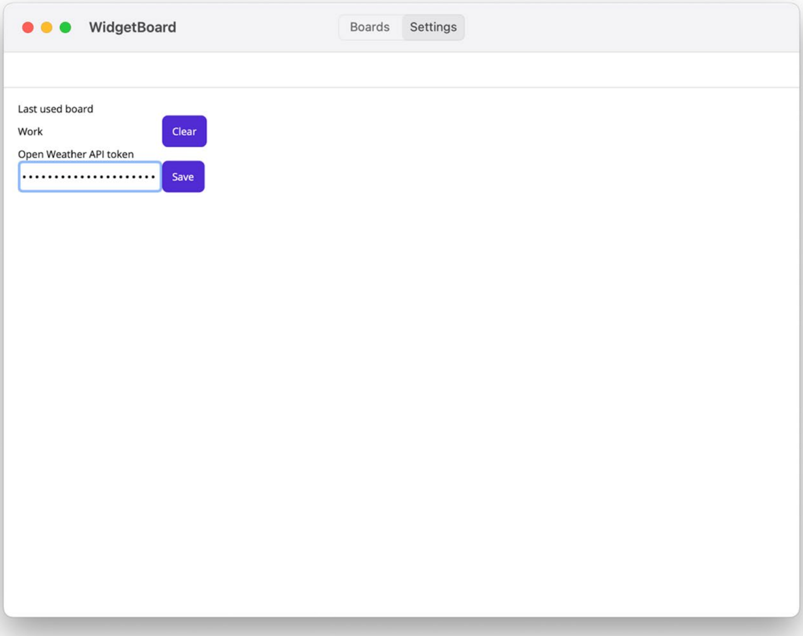

Figure 11-1. *Application running and showing an API key entered*

Then you can open a board and add a weather widget; you can see the result in Figure 11-2.

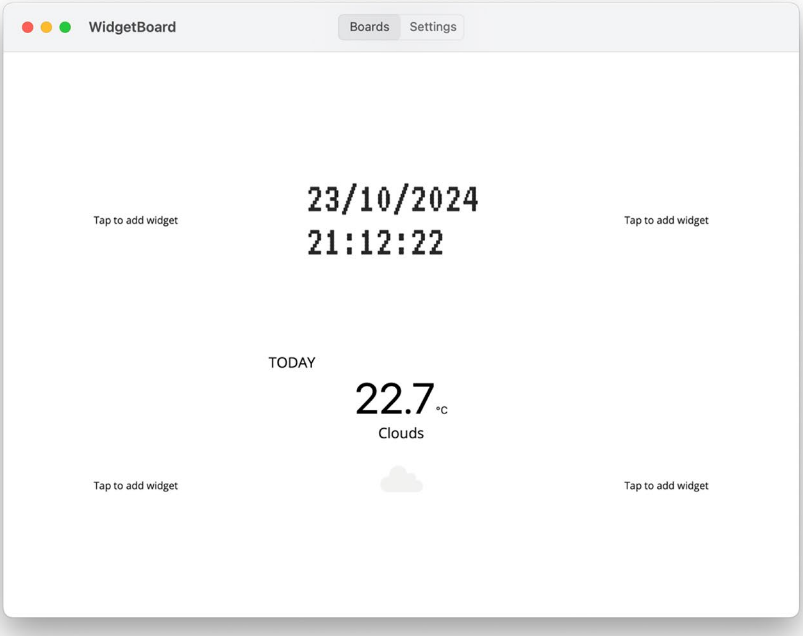

Figure 11-2. *Application running and showing your weather widget rendering correctly*

This works fine provided you have a good network connection. The moment you have a slow connection or even no connection, you will notice that things don't load quite as expected. In fact, you will likely observe a crash. You knew this could happen based on your earlier investigation into the things you need to consider when handling remote data. Let's now apply some techniques to handle these scenarios.

Adding Some State

The first thing you want to do is to consider the different possible states that your process can be in. There are three key scenarios that you need to handle and provide visual feedback to your users on:

1. The widget is loading the data.
2. The widget has the data.
3. The widget has encountered an issue loading the data.

Let's handle these three scenarios.

First, create an enum that will represent the above scenarios. You can add this to the root of the project.

```
namespace WidgetBoard;

public enum State
{
    None = 0,
    Loading = 1,
    Loaded = 2,
    Error = 3
}
```

You also want to modify your loading code in the view model to make use of this new `State`, with the changes in **bold**.

```
private async Task LoadWeatherForecast()
{
    var apiKey = await this.secureStorage.GetAsync("OpenWeather
    ApiToken");
```

```
    if (apiKey is null)
    {
        return;
    }

    try
    {
        State = State.Loading;

        var forecast = await weatherForecastService.
        GetForecast(20.798363, -156.331924, apiKey);

        if (forecast?.Main is null)
        {
            State = State.Error;
            return;
        }

        Temperature = forecast.Main.Temperature;
        Weather = forecast.Weather.First().Main;
        IconUrl = forecast.Weather.First().IconUrl;

        State = State.Loaded;
    }
    catch (Exception)
    {
        State = State.Error;
    }
}
```

The example above hasn't added any extra logging, but I would strongly advise that inside the `catch` statement, you log errors out so that you can investigate the reason for the error.

And you also need to add the State property and backing field.

```
private State state;
public State State
{
    get => state;
    set => SetProperty(ref state, value);
}
```

Converting the State to UI

This section may well deserve a more prominent setting; however, to allow the content to flow through this book, I opted to only expose parts based on the context of the topics you are learning as you build your application. Quite often in .NET MAUI, there are scenarios where you wish to bind a piece of data to the UI but that data type does not match the desired type in the UI. To avoid having to add additional properties and potentially adding view-related information into your view models, you can make use of a concept called *converters*. A converter enables you to define how a specific data type can be converted from its type to another type. I always find the best way to cover something like this is to see it in action, so let's create a converter to convert from your new State enum above into a bool value ready for binding to the IsVisible property in your view.

Add a new folder and call it Converters and then add a new class file and call it IsEqualToStateConverter.cs and then you can add the following contents:

```
using System.Globalization;

namespace WidgetBoard.Converters;

public class IsEqualToStateConverter : IValueConverter
{
    public State State { get; set; }
```

```
    public object? Convert(object? value, Type targetType,
    object? parameter, CultureInfo culture)
    {
        if (value is State state)
        {
            return state == State;
        }
        return value;
    }

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

The `IValueConverter` interface allows you to define how a value passed in can be converted. Implementations of this interface are for use within a binding using the `Converter` property. Let's proceed to learn how a converter can be used in a view.

Displaying the Loading State

It is worth noting that at times data can be loaded very quickly and the act of showing a spinner can provide a negative experience if it flashes very quickly. Of course, it is impossible to know which calls will take longer than others as there are so many factors which can affect the network. At times like this, I like to make sure that there is always a minimum amount of time that you display the spinner so that there isn't this weird flash to the user. Open the *WeatherWidgetView.xaml* file and make the following changes in **bold**:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentView
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:WidgetBoard.ViewModels"
    xmlns:converters="clr-namespace:WidgetBoard.Converters"
    x:Class="WidgetBoard.Views.WeatherWidgetView"
    x:DataType="viewmodels:WeatherWidgetViewModel">
    <ContentView.Resources>
        <converters:IsEqualToStateConverter
            x:Key="IsLoadingConverter"
            State="Loading" />
    </ContentView.Resources>
    <VerticalStackLayout>
        <Label
            Text="Today"
            FontSize="20"
            VerticalOptions="Center"
            HorizontalOptions="Start"
            TextTransform="Uppercase" />
        <!-- Loading -->
        <VerticalStackLayout
            IsVisible="{Binding State,
            Converter={StaticResource IsLoadingConverter}}">
            <ActivityIndicator
                IsRunning="{Binding State, Converter=
                {StaticResource IsLoadingConverter}}" />
            <Label
                Text="Loading weather data" />
        </VerticalStackLayout>
    </VerticalStackLayout>
</ContentView>
```

Displaying the Loaded State

In order to handle the error state, you need to add another instance of your `IsEqualToStateConverter`, this time with the `State` property set to `Loaded`.

```
<converters:IsEqualToStateConverter
    x:Key="HasLoadedConverter"
    State="Loaded" />
```

You can then use this converter in a binding to show/hide the following UI; note that this will replace the original contents of the widget:

```
<!-- Loaded -->
<VerticalStackLayout
    IsVisible="{Binding State, Converter={StaticResource
    HasLoadedConverter}}">
    <Label
        VerticalOptions="Center"
        HorizontalOptions="Center">
        <Label.FormattedText>
            <FormattedString>
                <Span
                    Text="{Binding Temperature,
                    StringFormat='{0:F1}'}"
                    FontSize="60"/>
                <Span
                    Text="°C" />
            </FormattedString>
        </Label.FormattedText>
    </Label>
```

```
    <Label
        Text="{Binding Weather}"
        FontSize="20"
        VerticalOptions="Center"
        HorizontalOptions="Center" />
    <Image
        Source="{Binding IconUrl}"
        WidthRequest="100"
        HeightRequest="100"/>
</VerticalStackLayout>
```

Displaying the Error State

In order to handle the error state, you need to add another instance of your `IsEqualToStateConverter`, this time with the `State` property set to `Error`.

```
<converters:IsEqualToStateConverter
    x:Key="HasErrorConverter"
    State="Error" />
```

You can then use this converter in a binding to show/hide the following UI:

```
<!-- Error -->
<VerticalStackLayout
    IsVisible="{Binding State, Converter={StaticResource
    HasErrorConverter}}">
    <Label
        Text="Unable to load weather data" />
    <Button
        Text="Retry"
        Command="{Binding LoadWeatherCommand}" />
</VerticalStackLayout>
```

You may have noticed that you have added a `Button` and bound its command to the view model. You need to add this to your view model if you wish to compile and run the application. The aim of the `Button` is to allow the user to request a retry of loading the weather information if the `Error` state is being shown.

Inside your `WeatherWidgetViewModel.cs` file, you need to make the following change:

```
public ICommand LoadWeatherCommand { get; }
```

Then you need to update the constructor with the changes in bold:

```
public WeatherWidgetViewModel(IWeatherForecastService
weatherForecastService, ISecureStorage secureStorage)
{
    this.weatherForecastService = weatherForecastService;
    this.secureStorage = secureStorage;

    LoadWeatherCommand = new Command(async () => await
    LoadWeatherForecast());
    Task.Run(async () => await LoadWeatherForecast());
}
```

This means that when a load fails for whatever reason, the user will have the option to press the retry button and the widget will attempt to load the weather details again. It will walk through the states you added, so the UI will show the different UI options to the user as this happens.

This type of failure handling is considered manual. There are ways to automatically handle retries through a package called Polly.

Network Resilience Handling

Polly is a fantastic package that started off as a community-based implementation and has now been adopted into the .NET ecosystem through the `Microsoft.Extensions.Http.Resilience` package. The code that was added earlier to load data from the Open Weather API isn't the most complex, but if you decided to add in the ability to handle connection retries if a request fails and increase the delay between retry attempts, I am sure you can imagine how complex it could become. We are going to add such a feature in just a few lines with this new NuGet package.

Let's go ahead and add the *Microsoft.Extensions.Http.Resilience* NuGet package and then take a look at how to use it.

- Right-click the *WidgetBoard* solution.
- Select **Manage NuGet Packages**.
- Search for *Microsoft.Extensions.Http.Resilience*.
- Select the correct package.
- Click **Add Package**.

Now you can open the MauiProgram.cs file and make the following changes (in **bold**):

```
builder.Services.AddHttpClient<WeatherForecastService>()
    .AddStandardResilienceHandler(static options =>
    {
        options.Retry = new HttpRetryStrategyOptions
        {
            BackoffType = DelayBackoffType.Exponential,
            MaxRetryAttempts = 3,
            UseJitter = true,
            Delay = TimeSpan.FromSeconds(2)
        };
    });
```

The above code adds a new retry strategy that will

- Retry a maximum of three attempts.
- The first delay between retries will be two seconds, and the delay will exponentially grow on each retry attempt.
- UseJitter means there will be a random factor to the delay.

That is all we need to add in a layer of resilience to the application. Rules like this can be especially useful in the mobile world! This only introduces what Polly can offer; if you want to learn more, then I would highly recommend reading the documentation on all that it offers (`https://github.com/app-vnext/polly`).

Simplifying Web Service Access

The previous sections covered how you can interact directly with a web service at the most basic level. It requires a bit of setup, but thankfully in your scenario, this wasn't too complicated. Some web services can require a lot more setup or even return a lot more data.

When building your applications, the aim is to write as little code as possible as it reduces the amount of code you need to maintain. This statement isn't advocating for writing shortened code that can be difficult for a human to understand but instead stating that you want to focus on the details that are core to the application that you are building and not things like consuming a web service. Sure, you want to know that you are, but having to write the underlying bits through the use of `HttpClient` can become cumbersome. Thankfully there are packages out there that can help you!

Prebuilt Libraries

I first recommend that you investigate whether the web service provider also provides a client library to make the consumption easier. Quite often providers supply a library, especially when there is a layer of authentication required. There are no official client libraries for the Open Weather API; however, there are a number of NuGet packages that provide some support for using the API.

Code Generation Libraries

If no client library is available, you can look to using an auto generation package to reduce the amount of code you need to write. Refit is a fantastic package for this purpose. It allows you to define an interface representing the web service call, and then Refit will do the rest.

So why didn't I just start here? In a new project, you probably would do so, but I always strongly feel that you need to gain an understanding of what packages like Refit are doing before you really start to use them. This can be invaluable when things go wrong and you have to debug exactly what and why things are going wrong!

Adding the Refit NuGet Package

Let's go ahead and add the *Refit.HttpClientFactory* NuGet package and then take a look at how to use it.

- Right-click the *WidgetBoard* solution.
- Select **Manage NuGet Packages**.
- Search for *Refit.HttpClientFactory*.
- Select the correct package.
- Click **Add Package**.

Now that you have the NuGet package installed, you can use it. Open your `IWeatherForecastService.cs` file and make the following modifications shown in **bold**:

```
using Refit;

namespace WidgetBoard.Communications;

public interface IWeatherForecastService
{
    [Get("/weather?lat={latitude}&lon={longitude}&units=metric&
    appid={apiKey}")]
    Task<Forecast?> GetForecast(double latitude, double
    longitude, string apiKey);
}
```

The fantastic part of the above code is that you do not need to write the implementation. Refit uses source code generators to do it for you! In fact, it means you can delete your `WeatherForecastService` class as it is no longer required.

The final change you are required to make is to change how you register the `IWeatherForecastService` with your `MauiAppBuilder` in the `MauiProgram.cs` file. Open it up and make the following changes.

First, add the `using` statement.

```
using Refit;
```

Then replace

```
builder.Services.AddSingleton<IWeatherForecastService,
WeatherForecastService>();
```

with

```
builder.Services
    .AddRefitClient<IWeatherForecastService>()
    .ConfigureHttpClient(c => c.BaseAddress = new Uri("https://
    api.openweathermap.org/data/2.5"));
```

This new line of code makes use of the Refit extension methods that enable you to consume an implementation of `IWeatherForecastService` whenever you register a dependency on that interface. It is worth reiterating that the implementation for the `IWeatherForecastService` is automatically generated for you through the Refit package. For further reading on this package, I thoroughly recommend their website at `https://reactiveui.github.io/refit/`.

If you run up the application, you will see the same result as Figure 11-2 – the weather widgets the weather as expected.

Further Reading

You have added some complexities into your application in order to handle the scenario when web service access doesn't load as expected. There is a really great library that can really help to reduce the amount of code you need to write around these parts.

StateContainer from CommunityToolkit.Maui

You had to build in converters and apply `IsVisible` bindings to control which view is being displayed when your widget is in a specific state. The `StateContainer` reduces that overhead so you "just" need to define the states and the views for those states.

If you love to write less code, I recommend checking out the Microsoft documentation at `https://learn.microsoft.com/dotnet/communitytoolkit/maui/layouts/statecontainer`.

Summary

In this chapter, you have

- Learned about remote data
- Learned how you can interact with it
- Covered the common considerations
- Looked a concrete example with the Open Weather API
- Built your own implementation to consume the Open Weather API
- Covered how to consume the data returned
- Talked through scenarios where things can go wrong
- Provided implementations to handle those scenarios
- Looked at how you can reduce the complexity of your implementation with Refit
- Added in your Weather Widget

In the next chapter, you will

- Learn about permissions on the various platforms and how to request them
- Learn how to use the Geolocation API
- Cover how to write your own platform-specific interaction when necessary
- Discover how to tweak the UI based on the platform on which your application is running
- Learn to tweak the UI through the use of the handler architecture

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch11`.

Extra Assignment

There are so many possibilities for accessing remote data in your application! Here are some extra widgets I would like you to consider creating.

TODO Widget

The go-to example application to build in tutorials is a TODO application. I would like you to expand upon this idea and add a TodoWidget into your application. There are several TODO APIs that you could utilize to do this. Do you have a favorite TODO service that you use? I personally like the Microsoft TODO option. There is some good documentation over on the Microsoft pages to help get you started at `https://learn.microsoft.com/graph/todo-concept-overview`.

Quote of the Day Widget

I know I certainly like to be inspired with a feel-good quote. Why don't you consider building a widget to refresh daily and show you a quote of the day?

The *They Said So Quotes API* offers a good API for doing this exact job with the documentation hosted at `https://quotes.rest/`.

The other concept that you will need to consider is how to trigger your `Scheduler` class to trigger the refresh at midnight.

NASA Space Image of the Day Widget

I love some of the images that come from NASA. It is so cool to be able to see into the reaches of space! Quite handily, they have a decent set of APIs that can enable you to build a widget and show off these images! The documentation on the NASA website really is great and should be able to guide you through the process of accessing the data you need. The NASA API documentation can be found at https://api.nasa.gov/.

I really can't wait to see these widgets in action!

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch11-extra.

CHAPTER 12

Getting Specific

Abstract

In this chapter, you will be learning about .NET MAUI Essentials and how it enables you to access platform-specific APIs without having to worry about any of the platform-specific complexities. Two concrete examples are requesting permissions on each platform and accessing the device's geolocation information. You will explore what is required if you really do need to interact with platform-specific APIs that have not been abstracted for you. Finally, you will cover multiple techniques, concepts, and architectures that enable you to tweak the UI and behavior of your applications based on the platforms they are running on.

.NET MAUI Essentials

In the previous chapter, you created a weather widget. You did not finish the job, though, as it currently only loads the weather for Maui, Hawaii. I don't know about you, but I am not lucky enough to live there! In this section, you will discover what the current device's location is in terms of longitude and latitude, and you will then send that information up to the Open Weather API for a much more accurate weather summary of the user's current location.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_12

In order to achieve this, you need an understanding of two key concepts: the permissions system of each operating system and how to access the APIs specific to GPS coordinates. Thankfully .NET MAUI has you covered for both scenarios, but you do need to be aware of how they work and any platform-specific differences. Let's take a look at each to get a better understanding.

Permissions

A common theme I have been discussing in this book is how .NET MAUI does a lot of the heavy lifting when it comes to dealing with each supported platform. This continues with permissions because .NET MAUI abstracts a large number of permissions.

It is worth noting that every operating system is different. Not all require permissions for certain features. Refer to the Microsoft documentation on what .NET MAUI supports and what is required for each platform at `https://learn.microsoft.com/dotnet/maui/platform-integration/appmodel/permissions-available-permissions`.

There are two key methods that enable you to interact with the permission system in .NET MAUI.

Checking the Status of a Permission

In order to check whether the user has already granted permission for your application to use a specific feature, you can use the `CheckStatusAsync` method on the `Permissions` class. For your weather widget, you need access to the device's geolocation information. You have two options in terms of the permission to use:

- `LocationWhenInUse`: This only allows the application to access the geolocation information while the app is open in the foreground.

- `LocationAlways`: This allows the application to also access the geolocation information even when the app is backgrounded. This can be particularly useful for exercise tracking applications that need to monitor the user's movement.

You only need the `LocationWhenInUse` option for your application.

```
PermissionStatus status = await Permissions.
CheckStatusAsync<Permissions.LocationWhenInUse>();
```

It is recommended that you check the status of the permission before requesting it to gain an understanding of whether the user has been asked before. On iOS, you are only allowed to ask once, and then you are required to prompt the user to go to the Settings app and enable permission if they wish to change their mind. Sadly, Android provides a different approach and will return a status of `Denied` even if the user has not been prompted before. In this scenario, you are then recommended to call `ShouldShowRationale` to check whether the user really has been prompted.

The possible values for `PermissionStatus` are as follows:

- `Unknown`: The permission is in an unknown state, or on iOS, the user has never been prompted.
- `Denied`: The user denied the permission request, or on Android, the user *might* not have been prompted.
- `Disabled`: The feature is disabled on the device.
- `Granted`: The user granted permission or it is automatically granted.
- `Restricted`: In a restricted state.

Requesting Permission

Once you have confirmed that the user has not been prompted with a permission request, you can proceed to prompting them by using the `Permissions.RequestAsync` method along with the specific permission to request. In your example, this will be the `LocationWhenInUse` permission.

```
PermissionStatus status = await Permissions.
RequestAsync<Permissions.LocationWhenInUse>();
```

It is worth noting that the `RequestAsync` method needs to be run on the main or UI thread. This is needed because it can result in presenting the built-in system UI in order to ask the user if they wish to give permission. Therefore, whenever you call `Permissions.RequestAsync`, you must make sure your code is already running on the main thread with the `MainThread.IsMainThread` property, or you can dispatch out to the main thread with the `MainThread.InvokeOnMainThreadAsync` method.

It is considered best practice to only prompt the user for permission to use a specific feature when they first try to use that feature. This helps to provide context to the user around why the permission is being requested. You may also find that the different platform providers (e.g., Apple, Google, and Microsoft) have different rules they apply when reviewing and approving the applications you submit to their stores. For this, I recommend working with the most restrictive rules to save yourself pain and effort.

Some further points that I want to highlight are as follows:

1. When considering the user experience, it is always good to add a page into your application that explains in enough detail why certain permissions are needed. Once the user has had the chance to read this, only then request the permission and let the user agree.

2. Always make sure to not only test the happy path – the user has granted permission. You should always consider what will happen if permission has not been granted.

3. This partially fits into point 3, but also consider that certain features might be entirely disabled on a device (e.g., location services). Make sure that your application behaves as expected in these scenarios.

Handling Permissions in Your Application

The following section of code comes recommended from the Microsoft documentation site at `https://learn.microsoft.com/dotnet/maui/platform-integration/appmodel/permissions?#example`. It has been included and left unchanged as it helps to really highlight the differences between platforms.

First, create the new folder and class for this new piece of functionality. Call the folder `Services`. Add a new interface file and call it `ILocationService.cs` under the `Services` folder. The contents of this new interface should be updated to the following:

```
namespace WidgetBoard.Services;

public interface ILocationService
{
    Task<Location?> GetLocationAsync();
}
```

This is the definition of what a location service implementation will provide: an asynchronous method that will ultimately return a `Location` object.

Next, create an implementation. Add a new class file under the Services folder and call it LocationService.cs. Modify the initial contents to the following:

```
namespace WidgetBoard.Services;

public class LocationService : ILocationService
{
}
```

Now that you have a blank class, you can add the method for handling permission requests ready for use.

```
private async Task<PermissionStatus>
CheckAndRequestLocationPermission()
{
    PermissionStatus status = await Permissions.
    CheckStatusAsync<Permissions.LocationWhenInUse>();
    if (status == PermissionStatus.Granted)
    {
        return status;
    }

    if (status == PermissionStatus.Denied && DeviceInfo.
    Platform == DevicePlatform.iOS)
    {
        // Prompt the user to turn on in settings
        // On iOS once a permission has been denied it may not
        be requested again from the application
        return status;
    }
```

```
    if (Permissions.ShouldShowRationale<Permissions.
    LocationWhenInUse>())
    {
        // Prompt the user with additional information as to
        why the permission is needed
    }

    status = await Permissions.RequestAsync<Permissions.
    LocationWhenInUse>();

    return status;
}
```

Now that you have added the ability to request the user's permission to use the geolocation APIs on the device, you can proceed to using it.

Using the Geolocation API

.NET MAUI provides the ability to access each platform's geolocation APIs in order to retrieve a longitude and latitude representing where in the world the device running the application is currently located. Full details of what the API provides can be found at https://learn.microsoft.com/dotnet/maui/platform-integration/device/geolocation.

Registering the Geolocation Service

Open the `MauiProgram.cs` file and register the geolocation implementation so that you can use it via the dependency injection layer. You need to add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton(Geolocation.Default);
```

Using the Geolocation Service

This now means that you can add a dependency on the IGeolocation interface and wherever .NET MAUI provides you with an instance. Let's use the IGeolocation implementation in your LocationService.cs file. There are a few modifications you need to make, so I will walk through each one.

Add a field for the IGeolocation implementation in the root of the class.

```
private readonly IGeolocation geolocation;
```

Assign the IGeolocation implementation in the constructor.

```
public LocationService(IGeolocation geolocation)
{
    this.geolocation = geolocation;
}
```

Provide the method to return a Location object.

```
public async Task<Location?> GetLocationAsync()
{
    return await MainThread.InvokeOnMainThreadAsync(async () =>
    {
        var status = await CheckAndRequestLocationPermission();
        if (status != PermissionStatus.Granted)
        {
            return null;
        }
        return await this.geolocation.GetLocationAsync();
    });
}
```

This implementation first makes sure that you are running on the main thread, which is required for location-based access. Then it calls your permission handling method, and if the app has permission, it calls the `IGeolocation` implementation and returns the resulting `Location` object. Now you are ready to make use of the `LocationService`.

Registering the LocationService

Open the `MauiProgram.cs` file and register the `LocationService` implementation so that you can use it via the dependency injection layer. You need to add the following line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton<ILocationService,
LocationService>();
```

Using the ILocationService

Let's use the `ILocationService` implementation in your `WeatherWidgetViewModel.cs` file. There are a few modifications you need to make, so I will walk through each one

Add a field for the `ILocationService` implementation in the root of the class.

```
private readonly ILocationService locationService;
```

Assign the `ILocationService` implementation in the constructor; changes are in **bold**.

```
public WeatherWidgetViewModel(
    IWeatherForecastService weatherForecastService,
    ISecureStorage secureStorage,
    ILocationService locationService)
```

```
{
    this.weatherForecastService = weatherForecastService;
    this.locationService = locationService;
    LoadWeatherCommand = new Command(async () => await
    LoadWeatherForecast());
}
```

Modify your `State` enum to include a new value so that you can handle when something goes wrong with permission access. Add a `PermissionError` value, as can be seen below in **bold**.

```
public enum State
{
    None = 0,
    Loading = 1,
    Loaded = 2,
    Error = 3,
    PermissionError = 4
}
```

Modify your `LoadWeatherForecast` method to call your new `ILocationService` implementation in order to find out the device's location and then use that to call the Open Weather API to find out the weather at the device's location. The changes are in **bold**.

```
private async Task LoadWeatherForecast()
{
    var apiKey = await this.secureStorage.GetAsync("OpenWeather
    ApiToken");

    if (apiKey is null)
    {
        return;
    }
```

```
    try
    {
        State = State.Loading;

        var location = await this.locationService.
        GetLocationAsync();
        if (location is null)
        {
            State = State.PermissionError;
            return;
        }

        var forecast = await weatherForecastService.
        GetForecast(location.Latitude, location.Longitude,
        apiKey);

        Temperature = forecast.Main.Temperature;
        Weather = forecast.Weather.First().Main;
        IconUrl = forecast.Weather.First().IconUrl;

        State = State.Loaded;
    }
    catch (Exception ex)
    {
        State = State.Error;
    }
}
```

You have introduced a few changes here, so let's break them down.

First, you are calling the `locationService` to get the device's location. If it returns null, it means the application does not have permission and you set the `State` to `PermissionError`.

If you have permission, you pass the device's current location into the `weatherForecastService.GetForecast` method.

Displaying Permission Errors to Your User

You have added the new state value and also assigned it in your view model when you either fail to retrieve the permission setting or the user has denied permission to the `LocationWhenInUse` feature. Now you can add in support into your UI to respond to this value and show something appropriate to the user. Open the `WeatherWidgetView.xaml` file and make the following modifications.

Add in the converter instance inside the `<ContentView.Resources>` tag.

```
<converters:IsEqualToStateConverter
    x:Key="HasPermissionErrorConverter"
    State="PermissionError" />
```

Then you can add a section that will render when the `State` property is equal to `PermissionError`. You should add this into the `WeatherWidgetView.xaml` file after the following section:

```
<!-- Error -->
<VerticalStackLayout
            IsVisible="{Binding State,
            Converter={StaticResource HasErrorConverter}}">
    ...
</VerticalStackLayout>
The section you want to add is as follows:
<!-- PermissionError -->
<VerticalStackLayout
    IsVisible="{Binding State, Converter={StaticResource
    HasPermissionErrorConverter}}">
    <Label
        Text="Unable to retrieve location data" />
```

```
    <Button
        Text="Retry"
        Command="{Binding LoadWeatherCommand}" />
</VerticalStackLayout>
```

Now that you have added all of the required bits of code to call into the Permissions and Geolocation APIs, you need to configure each of your supported platforms to enable the location permission.

Configuring Platform-Specific Components

This is where .NET MAUI stops holding your hand and requires you to do some work in the platform-specific folders. Many of the APIs that are provided by .NET MAUI, as detailed in this section of the documentation site at `https://learn.microsoft.com/dotnet/maui/platform-integration/`, have the potential to require some level of platform-specific setup. This will vary per platform. For example, for haptic support, only Android requires some setup, whereas for the Geolocation API, all platforms require some setup.

Thankfully .NET MAUI provides helpful exceptions and error messages if you miss any of the platform-specific setup, and they usually indicate the action required to fix the issue. Topics like this do make it imperative that you really test your application on each of the platforms you wish to support to verify that it behaves as expected.

Let's set up each platform so that your app can fully support accessing the device's current location.

Android

Android requires several permissions and features to be configured in order for your application to use the `LocationWhenInUse` permission. You can configure them inside the `Platforms/Android/MainApplication.cs` file, so open it and make the following additions in **bold**:

```
using Android.App;
using Android.Runtime;
[assembly: UsesPermission(Android.Manifest.Permission.
AccessCoarseLocation)]
[assembly: UsesPermission(Android.Manifest.Permission.
AccessFineLocation)]
[assembly: UsesFeature("android.hardware.location", Required
= false)]
[assembly: UsesFeature("android.hardware.location.gps",
Required = false)]
[assembly: UsesFeature("android.hardware.location.network",
Required = false)]
namespace WidgetBoard;
```

Note that the use of the `assembly` keyword requires that the attributes are applied at the assembly level and not on the class like the current `[Application]` attribute usage. For further reference on how to get started with geolocation, refer to the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/platform-integration/device/geolocation?tabs=android-get-started.

If you run the application on Android now, you will see that the first time you add a Weather widget onto a board, the system will present the following popup to the user asking them to allow permission for your application to use the location feature. Figure 12-1 shows the result of running your application on Android.

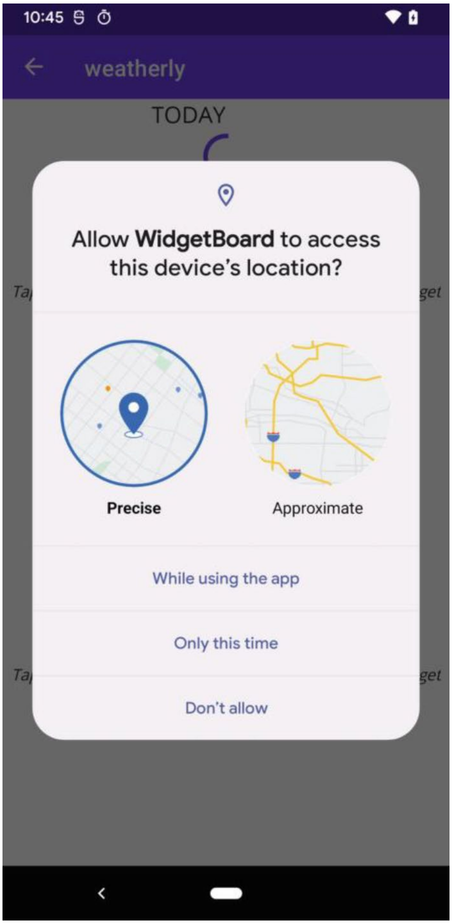

Figure 12-1. *The application running on Android showing the permission prompt when a weather widget is first added to a board*

iOS/Mac

Apple requires that you specify the reason your application wants to use the Geolocation feature in the process of defining that your application uses the feature. You can configure this by modifying the `Platforms/iOS/Info.plist` and `Platforms/MacCatalyst/Info.plist` files for iOS and

Mac Catalyst, respectively. Both files require the same change, so let's open them and add the following lines in. Note that I am opting to edit the files inside Visual Studio Code as I find it provides a better editing experience. There is a built-in editor inside Visual Studio, but I personally prefer to edit the XML directly. Add the following lines inside the `<dict>` element:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>In order to provide accurate weather
information.</string>
```

For further reference on how to get started with Geolocation, refer to the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/platform-integration/device/geolocation?tabs=ios-get-started.

If you run the application on iOS and macOS now, you will see that the first time you add a Weather widget onto a board, the system will present the following popup to the user asking them to allow permission for your application to use the location feature. Figure 12-2 shows the result of running the application on iOS.

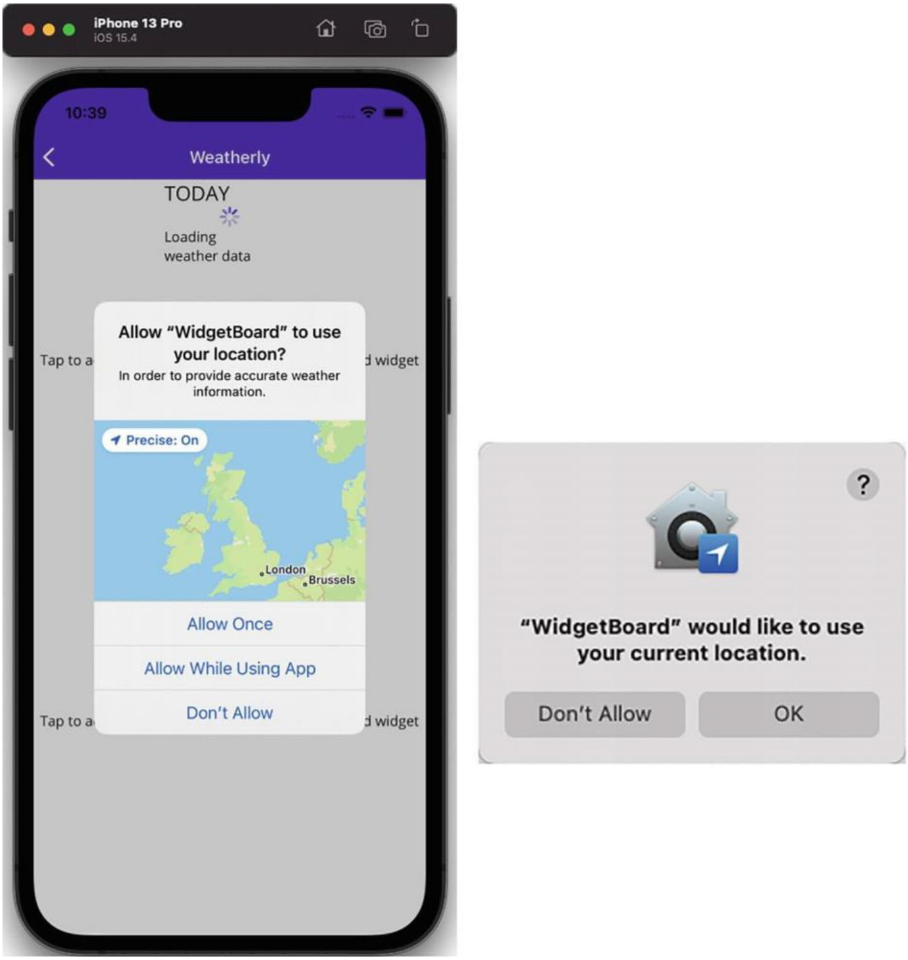

Figure 12-2. *The application running on iOS (left) and macOS (right) showing the permission prompt when a weather widget is first added to a board*

Windows

Windows applications have the concept of capabilities, and it is up to developers to declare which capabilities are required in their applications. In order to do so for your application, you need to modify the `Platforms/Windows/Package.appxmanifest` file. Note that I am opting to edit the files inside Visual Studio Code as I find it provides a better editing experience. Add the following line inside the `<Capabilities>` element:

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

For further reference on how to get started with Geolocation, refer to the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/platform-integration/device/geolocation?tabs=windows-get-started.

If you run the application on Windows now, you don't see a permission request popup. Figure 12-3 shows the result of running the application on Windows.

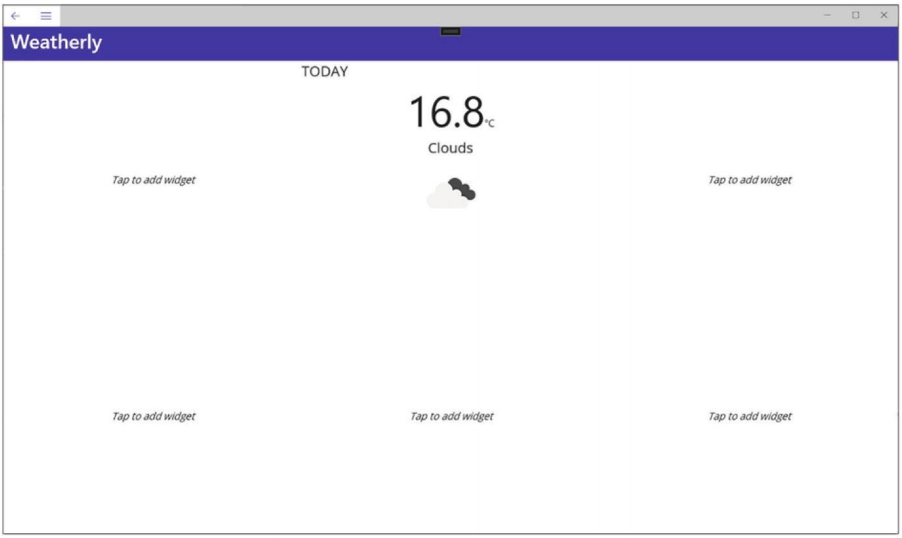

***Figure 12-3.** The application running on Windows showing the permission prompt when a weather widget is first added to a board*

Platform-Specific API Access

While .NET MAUI does provide you with a lot of functionality out of the box, there can be times when you need to write your own interaction with the platform-specific layer to achieve your goals. Whatever functionality can be achieved on a specific platform can also be achieved within a .NET MAUI application. You just might have to do the heavy lifting yourself. If your implementation is considered useful enough to other developers, you should propose the changes back to the .NET MAUI team.

There are two main concepts you can utilize when building platform-specific code in .NET MAUI. Let's take a look at each one through the simple example of building a `LocationService` that returns the longitude and latitude of the headquarters for each platform provider (e.g., Google, Apple, and Microsoft).

Platform-Specific Code with Compiler Directives

You will most likely come across a usage of the `#if` compiler directive when working on a .NET MAUI application. I am not a big fan of them, but I do accept that in some scenarios, they do provide value.

```
namespace WidgetBoard.Services;

public class PlatformLocationService : ILocationService
{
    public Task<Location?> GetLocationAsync()
    {
        Location? location;
#if ANDROID
        location = new Location(37.419857, -122.078827);
#elif WINDOWS
        location = new Location(47.639722, -122.128333);
```

```
#else
        location = new Location(37.334722, -122.008889);
#endif
        return Task.FromResult(location);
    }
}
```

The above code will be compiled in different ways based on the target platform. The resulting compiled code for the Android platform looks as follows:

```
namespace WidgetBoard.Services;

public class PlatformLocationService : ILocationService
{
    public Task<Location?> GetLocationAsync()
    {
        Location? location;
        location = new Location(37.419857, -122.078827);
        return Task.FromResult(location);
    }
}
```

This means that only the code specific to the platform will be compiled and shipped to that platform.

This approach can work well in this scenario, but as soon as you need to use multiple classes or other platform-specific libraries, the code will become complex very quickly. In more complex scenarios, you can use the platform-specific folders created in your project for you.

Platform-Specific Code in Platform Folders

I briefly covered these folders in Chapter 2. Each platform has a folder, and the files inside each folder (e.g., `/Platforms/Android/`) will only be compiled for that platform when you are targeting it. In order to create the same `PlatformLocationService` from the previous section, you first need to create a partial class under the `Services` folder with the following contents:

```
namespace WidgetBoard.Services;

public partial class PlatformLocationService : ILocationService
{
}
```

The above code will not compile now because you haven't implemented `ILocationService`. This is expected until you add in your platform-specific implementations, so don't worry. You add the `partial` keyword because this is only a partial implementation. The platform-specific files and classes you will add shortly will complete this partial implementation.

Next, you need to create your Android platform-specific implementation. To do this, you add a new class file under the `/Platforms/Android/` folder and call it `PlatformLocationService.cs`, just like the one above. You want to modify its contents to the following:

```
namespace WidgetBoard.Services;

public partial class PlatformLocationService
{
    public Task<Location?> GetLocationAsync()
    {
        return Task.FromResult<Location?>(
           new Location(37.419857, -122.078827));
    }
}
```

This class will only be compiled when the Android platform is being targeted, and therefore, you get a very similar compiled output to the one in the "Platform-Specific Code with Compiler Directives" section. The key difference is that you don't need to add any of those unpleasant `#if` directives.

When building platform-specific implementations this way, the `namespace` of your partial classes must match! Otherwise, the compiler won't be able to build a single class.

We now need to add in the implementations for iOS, macOS, and Windows; rather than stepping through the same steps as the Android implementation above, we will mix and match the two approaches that we have just covered: compiler directives and platform-specific folders.

Open the *PlatformLocationService.cs* file in the *Services* folder and modify the contents to match the below (with changes in **bold**)

```
namespace WidgetBoard.Services;

public partial class MultiPlatformLocationService :
ILocationService
{
#if !ANDROID
    public Task<Location?> GetLocationAsync()
    {
        Location? location;
#if WINDOWS
        location = new Location(47.639722, -122.128333);
#elif MACCATALYST || IOS
        location = new Location(37.334722, -122.008889);
#else
        location = null;
```

```
#endif
        return Task.FromResult<Location?>(location);
    }
#endif
}
```

This implementation will only include the `GetLocationAsync` method from this file for iOS, macOS, and Windows. The Android implementation will include the `GetLocationAsync` method from the */Platforms/Android/PlatformLocationService.cs* file. Sometimes this approach can work well if some platforms behave in the same way and can share implementations.

Overriding the Platform-Specific UI

One fundamental part of .NET MAUI is in the fact that it utilizes the underlying platform controls to handle the rendering of our applications. This will result in our applications looking different on each of the platforms. In the majority of scenarios, this is considered a good thing because the application is in keeping with the platform's look and feel. At times, though, you will need to override some of the platform-specific rendering or even just to tweak how controls render in your application on a specific platform.

OnPlatform

A common example of needing to change control properties is around the sizing of text or spacing around controls (`Margin` or `Padding`). I always find that the final finishing touches to get an application feeling really slick and polished can result in needing to tweak details like this per platform. There are two main ways to achieve this, and they depend on whether you are a XAML- or C#-oriented UI builder. Let's look over both with an example.

OnPlatform Markup Extension

XAML, as mentioned, is not as feature rich in terms of what can be written and achieved. Therefore, additional functionality is provided by .NET MAUI to overcome these limitations. One such example is the OnPlatform markup extension. XAML markup extensions help enhance the power and flexibility of XAML by allowing element attributes to be set from a variety of sources.

You might decide that in your `ClockWidgetView.xaml` file the `FontSize` property is too large for iOS and Android and opt to change it only for those platforms. Let's take a look at the code and see how you can modify the property based on the platform the application is running on.

```
<?xml version="1.0" encoding="utf-8" ?>
<Label
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:WidgetBoard.ViewModels"
    x:Class="WidgetBoard.Views.ClockWidgetView"
    FontSize="80"
    VerticalOptions="Center"
    HorizontalOptions="Center"
    x:DataType="viewmodels:ClockWidgetViewModel"
    Text="{Binding Time}">
    <Label.BindingContext>
        <viewmodels:ClockWidgetViewModel />
    </Label.BindingContext>
</Label>
```

The code above shows that the `FontSize` property is currently fixed to a value of `80`. With the `OnPlatform` markup extension, you can change this value based on the platform the application is running on. The following code example shows how you can retain the default value of 80 and then override for the platforms that you wish:

```
FontSize="{OnPlatform Default=80, Android=25, iOS=30}"
```

The code example above states that all platforms will default to using a `FontSize` of `60` unless the application is running on Android and a value of `25` will be used or if the application is running on iOS and a value of `30` will be used.

Conditional Statements

If you had built your UI in C# or wanted to at least modify the `FontSize` property of a `Label` control in a similar way, you could write the following conditional C# statement:

```
public ClockWidgetView()
{
    if (DeviceInfo.Platform == DevicePlatform.Android)
    {
        FontSize = 25;
    }
    else if (DeviceInfo.Platform == DevicePlatform.iOS)
    {
        FontSize = 30;
    }
    else
    {
        FontSize = 60;
    }
}
```

For further information on using the `OnPlatform` markup extension and other possible markup extensions that enable the customization of your application, please refer to the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/xaml/markup-extensions/consume#onplatform-markup-extension.

There will be times when just overriding values like this is not enough. For the more complex scenarios, you need to consider an architecture that is completely new to .NET MAUI, and that is the handler architecture.

Handlers

Handlers are an area where .NET MAUI really shines! If you have come from a Xamarin.Forms background, you will appreciate the pain that custom renderers brought. If you don't have any Xamarin.Forms experience, you are very lucky! I won't dig down too deep into the details of the old approach as this is a book on .NET MAUI and not the past; however, I feel there is value in talking about the old issues and how they have been overcome by the new handler architecture.

In both Xamarin.Forms and .NET MAUI, we predominantly build our user interfaces with abstract controls: controls defined in the Microsoft namespace and not specifically any platform controls. These controls eventually need to be mapped down to the platform-specific layer. In the Xamarin.Forms days, you would have a custom renderer. The renderer would be responsible for knowing about the abstract control and also the platform-specific control and mapping property values and event handlers and such between the two. This is considered a tightly coupled design, meaning that it becomes really quite difficult to enhance the controls and their rendering. If you wanted to override a small amount of behavior, you would have to implement a full renderer responsible for mapping all properties/events. This was very painful!

In .NET MAUI, this concept of renderers has been entirely replaced with handlers. This new architecture provides some extra layers between the abstract controls in the .NET MAUI namespace and the underlying platform-specific controls being rendered in our applications. This is considered much more loosely coupled, mainly due to the fact that each control will implement a number of interfaces and it is the handler's responsibility to interact with the interface rather than the specific control. This has many benefits including the fact that multiple controls can all implement the same interface and ultimately rely on the same single handler. It also provides the ability to define smaller chunks of common functionality, and as you all know, smaller classes and files are much easier to read, follow, and ultimately maintain. Figure 12-4 shows how the abstract `Button` class in .NET MAUI is mapped to the specific controls on each platform.

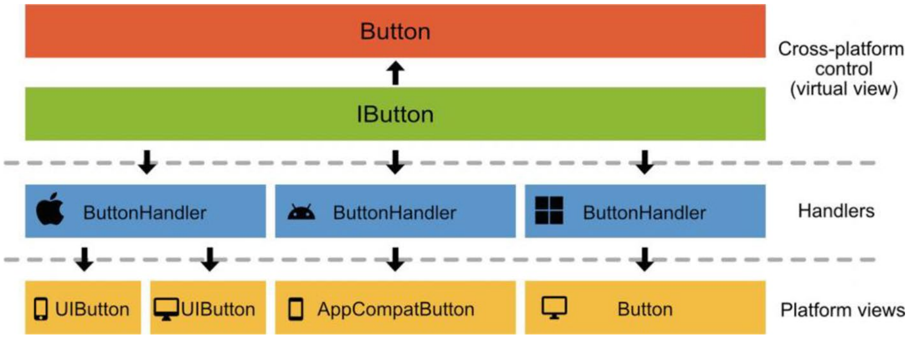

***Figure 12-4.** The handler architecture in .NET MAUI*

If you wish to create a new control that needs to map to platform-specific implementations, you should follow the pattern shown in Figure 12-4. Let's take the example of building a control to represent our `BoardLayout` control that we created in Chapter 7. We opted to build it using existing .NET MAUI controls, and therefore, we didn't need to implement any platform-specific components. We won't be building an actual control to support this, but we can look over the theory and examples to show how it could have been implemented. Figure 12-5 shows how the BoardLayout class would map onto platform-specific controls on each layer.

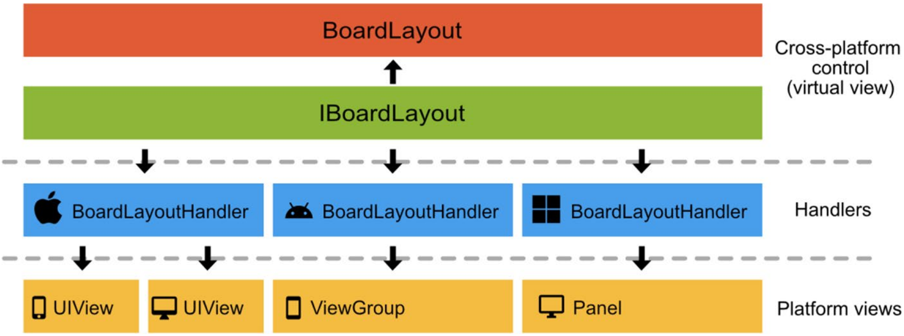

Figure 12-5. *The handler architecture representing the BoardLayout class in .NET MAUI*

You didn't take this approach in your scenario because there was no benefit. In fact, it would result in more code because you would need to map to each platform individually as well as implementing the layout logic in each of those platform-specific layers. This concept may sound like it will always cause more effort; however, in the situation of a Button, it makes sense because each platform already has a definition of what a button is and how it behaves.

Quite often as application developers, you will be using existing controls rather than building your own controls, so rather than needing to build everything you see in Figure 12-5, you can customize controls through the use of handlers.

Customizing Controls with Mappers

Mappers are key to the handler architecture. They define the actions that will be performed when either a property is changed or a command is sent between cross-platform controls and platform-specific views.

As an application developer, you have the ability to customize how these mappings work. Let's take a look at a concrete example – the `FixedDetailsPage` that was added earlier on in the book has three `Entry` controls to allow for user input. The following examples will show how to make the application select all text inside an `Entry` when the user focuses on it, apply this behavior to all `Entry` controls, and how to limit this behavior by mapping to specific implementations.

First, let's look at the platform-specific APIs that enable you to implement the functionality. Don't worry about adding code into the application just yet; we will look over the implementation details and then apply them later.

Select All Text on Android

The .NET MAUI `EntryHandler` on Android uses the `AppCompatEditText` widget. Handily this implementation provides the `SetSelectAllOnFocus` method which makes the Android implementation a one-liner.

```
handler.PlatformView.SetSelectAllOnFocus(true);
```

Select All Text on iOS/Mac Catalyst

The .NET MAUI `EntryHandler` on iOS and Mac Catalyst uses the `UITextField` control. There are two steps to selecting all text in our scenario; first, we would need to subscribe to the `EditingDidBegin` event, and then inside that event, we would need to perform the `selectAll` selector.

```
handler.PlatformView.EditingDidBegin += (s, e) =>
{
    handler.PlatformView.PerformSelector(new ObjCRuntime.
    Selector("selectAll"), null, 0.0f);
};
```

Select All Text on Windows

The .NET MAUI EntryHandler on iOS and Mac Catalyst uses the TextBox control. There are two steps to selecting all text on Windows: first, we would need to subscribe to the GotFocus event, and then inside that event, we would need to call SelectAll.

```
handler.PlatformView.GotFocus += (s, e) =>
{
    handler.PlatformView.SelectAll();
};
```

If you are familiar with any specific platform development (e.g., iOS development in Swift), the code to perform the selecting of all text should look familiar to doing it in that language (e.g. Swift), and that is because it is using the platform-specific APIs.

This covers how to perform the selecting of all text when an Entry gains focus; the final steps cover how to register this behavior to be performed.

Applying the Handler Globally

If you wanted to apply the above behavior globally to **all** Entry controls in your application, then you could add the following code into the CreateApp method in your *MauiProgram.cs* file:

```
Microsoft.Maui.Handlers.EntryHandler.Mapper.AppendToMapping(
"SelectAllText", (handler, view) =>
    {
#if ANDROID
    handler.PlatformView.SetSelectAllOnFocus(true);
#elif IOS || MACCATALYST
    handler.PlatformView.EditingDidBegin += (s, e) =>
```

```
    {
        handler.PlatformView.PerformSelector(new ObjCRuntime.
        Selector("selectAll"), null, 0.0f);
    };
#elif WINDOWS
    handler.PlatformView.GotFocus += (s, e) =>
    {
        handler.PlatformView.SelectAll();
    };
#endif
});
```

The result of the above will be to append our custom mapping that we are naming "SelectAllText" to the `EntryHandler`, which means all `Entry` controls will inherit this mapping.

Applying the Handler on a Single Page

Alternatively if you wish to only apply this custom behavior on the `BoardDetailsPage` screen of the application, then you can register the mapping in the constructor for that page. Open the BoardDetailsPage.xaml.cs file and modify the constructor to look as follows (changes in **bold**):

```
public BoardDetailsPage(BoardDetailsPageViewModel
boardDetailsPageViewModel)
{
    InitializeComponent();
    BindingContext = boardDetailsPageViewModel;
            Microsoft.Maui.Handlers.EntryHandler.Mapper.Appen
            dToMapping("SelectAllText", (handler, view) =>
```

```
    {
#if ANDROID
        handler.PlatformView.SetSelectAllOnFocus(true);
#elif IOS || MACCATALYST
        handler.PlatformView.EditingDidBegin += (s, e) =>
        {
            handler.PlatformView.PerformSelector(new
            ObjCRuntime.Selector("selectAll"), null, 0.0f);
        };
#elif WINDOWS
        handler.PlatformView.GotFocus += (s, e) =>
        {
            handler.PlatformView.SelectAll();
       };
#endif
    });
}
```

The result of the above will be to append our custom mapping that we are naming “SelectAllText” to the `EntryHandler`, which means all `Entry` controls on the `BoardDetailsPage` will inherit this mapping.

Applying the Handler to Specific Instances

If you wish to only modify the behavior for a subset of the Entry controls in your page/application, then you can do one of the following:

1. Subscribe to the `HandlerChanged` event from the Entry control and register the same code as above.
2. Create your own `Entry` subclass (let’s call it `MyEntry`), and then in the above code section, place an `if (view is MyEntry)` statement around the compiler directives.

Summary

In this chapter, you

- Learned about permissions on the various platforms and how to request them
- Learned how to use the Geolocation API
- Wrote your own platform-specific interaction when necessary
- Discovered how to tweak the UI based on the platform upon which your application is running
- Further tweaked the UI through the use of the handler architecture

In the next chapter, you will

- Learn what testing is and why it is important
- Cover what unit testing is and how you can apply it to a .NET MAUI application
- Learn what snapshot testing is and how you can implement it
- Gain an understanding of device tests and how you can apply them to your applications
- Look to the future for yet more testing goodness

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch12`.

Extra Assignment

You have only scratched the surface on the platform integration APIs that .NET MAUI offers you. I would love for you to look over the other possible APIs and build your own widgets that would benefit from them. The documentation for the platform integration APIs can be found at `https://learn.microsoft.com/dotnet/maui/platform-integration/`.

Barometer Widget

You can make use of the Barometer API in order to report the ambient air pressure back to the user. In fact, this might be a good addition to the Weather widget rather than a whole new widget. The documentation for this API can be found at `https://learn.microsoft.com/dotnet/maui/platform-integration/device/sensors?#barometer`.

Geocoding Lookup

I am reluctant to enable permissions like location access to apps I don't believe really need them. Perhaps you can enhance your Weather widget to allow the user to supply their nearest city, town, or postal code and then use the Geocoding API to reverse lookup the longitude and latitude information required for the Open Weather API. The documentation for the Geocoding API can be found at `https://learn.microsoft.com/dotnet/maui/platform-integration/device/geocoding`.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch12-extra`.

CHAPTER 13

Testing

Abstract

Testing is such an important part of the software development process; it enables you to verify that what you have delivered is what was required and also validate that the software behaves correctly. It also provides the safety net of catching regressions in the products that you build.

There are many different approaches for designing and writing tests and where they fit into the software development process. This chapter is not intended to provide full insight into those approaches, but it will expose you to various methods of testing a .NET MAUI application, why they can be beneficial, and pique your interest in learning to use them in more depth.

Unit Testing

Unit testing is the process of ensuring that small units, typically a method or class, of an application meet their design and behave as intended. One big benefit of testing such a small unit of the code is that it makes it easier for you to identify where issues may lie or creep in as part of regression. I have worked on many legacy systems throughout my career where the teams neglected to apply unit testing, and the experience when trying to identify the cause of a bug in a large system really can be costly in terms of time and money.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_13

Despite unit testing featuring near the end of this book, it is a concept that should be adopted early in the development process. Unit testing can aid in the design and building of code that is easier to read and maintain because it forces you to expose these small units of functionality and ultimately follow SOLID principles.

Unit testing itself will not catch all bugs in the system and should not be relied upon as a sole means of testing your applications. When used in combination with other forms of testing such as integration, functional, or end-to-end testing, you can build up confidence that your application is stable and delivers what is required.

Let's see how to implement unit testing with .NET MAUI.

Unit Testing in .NET MAUI

.NET MAUI applications are, as the name suggests, .NET-based projects, meaning that any of the existing .NET-based unit testing frameworks can be used.

As it currently stands, the default .NET MAUI project is not compatible with a unit test project. I will cover how to solve this in the "Adding a Unit Test Project to Your Solution" section.

There are three well-known frameworks that come with template support in Visual Studio, meaning you can create them with the File ➤ Add New Project option. The three frameworks are listed below.

xUnit

xUnit (`https://xunit.net`) appears to be the choice of the .NET MAUI team. One main reason for this is likely the support around being able to run xUnit-based unit tests on actual devices, meaning you can test device-specific implementations.

NUnit

NUnit (https://nunit.org) is an old favorite of mine. I have used it on so many projects in the past! It has some great features like being able to run the same test case with multiple sets of data to reduce the amount of testing code you need to write and ultimately maintain.

MSTest

MSTest is a testing framework that is built and supplied by Microsoft. It doesn't appear as feature rich as NUnit or xUnit, but it still does a great job (https://learn.microsoft.com/dotnet/core/testing/unit-testing-with-mstest).

Your Chosen Testing Framework

We will be using xUnit for this book mainly due to the benefits it brings with being able to also run the unit tests on devices.

Tests in xUnit are decorated with the `[Fact]` attribute with the expectation that as the author of the test methods, you will name them in a way that defines a fact which the test will prove to be true.

Most of the test frameworks are quite similar and tend to differ in terms of keywords when identifying tests. Go with whatever testing framework you are most comfortable with. If you do not have much experience with any, perhaps experiment with each to see which gives you the best experience. At the end of the day, you will be building and maintaining these tests so it needs to benefit you and your team.

Adding Your Own Unit Tests

There are some steps that you need to follow in order to make sure that you can unit test your .NET MAUI application – I should add that this is only if you wish to unit test the .NET MAUI application project directly; if you have class library projects with the code you wish to unit test, then this can be done without the following changes. Let's add a test project to the solution and then make the necessary changes.

Adding a Unit Test Project to Your Solution

1. Click the **File** menu.
2. Click **Add**.
3. Click **New Project**.
4. Enter *Test* in the **Search for templates** box.
   Figure 13-1 shows the Add a new project dialog in Visual Studio.

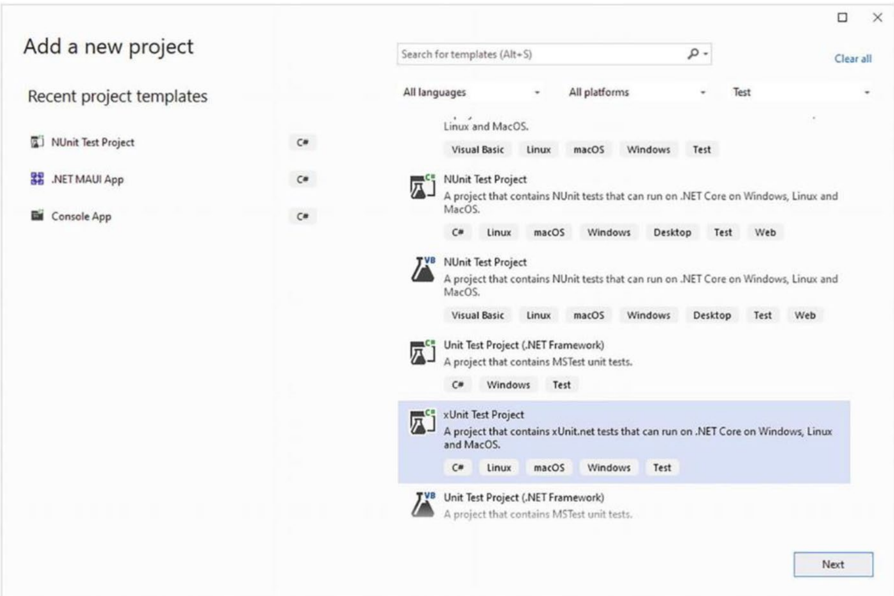

Figure 13-1. *Add a new project dialog in Visual Studio*

5. Select **xUnit Test Project**.

6. Click **Next**.

7. Enter a name for the project. I opted for *WidgetBoard.Tests* and find that appending .Tests or .UnitTests provides a common way to distinguish between application and test projects. This is also a common naming convention that simplifies searching for all unit test projects when running in a CI pipeline. I will cover this in more detail in Chapter 15.

8. Click **Next**.

9. Select the framework. The default should be fine; just make sure it matches the target version of the .NET MAUI application project.

10. Click **Create**.

Modify Your Application Project to Target net9.0

Sadly, the current .NET MAUI project template does not include the net9.0 target framework, meaning that it is not initially compatible with a standard unit test project. In order to correct this, you can manually add the net9.0 target framework. Open the `WidgetBoard/WidgetBoard.csproj` file in Visual Studio Code or your favorite text editor and make the following changes.

Modify the first `TargetFrameworks` element to include `net9.0` (changes in **bold**):

```
<TargetFrameworks>net9.0;net9.0-android;net9.0-ios;net9.0-
maccatalyst</TargetFrameworks>
```

Add a `Condition` attribute to the `OutputType` element (changes in **bold**):

```
<OutputType Condition="'$(TargetFramework)' != 'net9.0'">Exe
</OutputType>
```

Without this second change, you will see a compilation error reporting that *error CS5001: Program does not contain a static 'Main' method suitable for an entry point*. This is due to the fact that you are building an application and .NET applications expect to have a static `Main` method as the entry point to the application. The `OutputType` for .NET MAUI applications **must** be `Exe`, which might feel slightly confusing as you rarely end up with an exe file that will be delivered.

If you are building against a newer version of .NET MAUI, you can replace `net9.0` with the version you are using, such as `net10.0`.

Adding a Reference to the Project to Test

Now you need to add a reference from your test project onto the main application project.

1. Right-click **WidgetBoard.Tests**.
2. Click **Add**.
3. Click **Project Reference**.
4. Select **WidgetBoard** from the list. Figure 13-2 shows the Reference Manager dialog in Visual Studio.

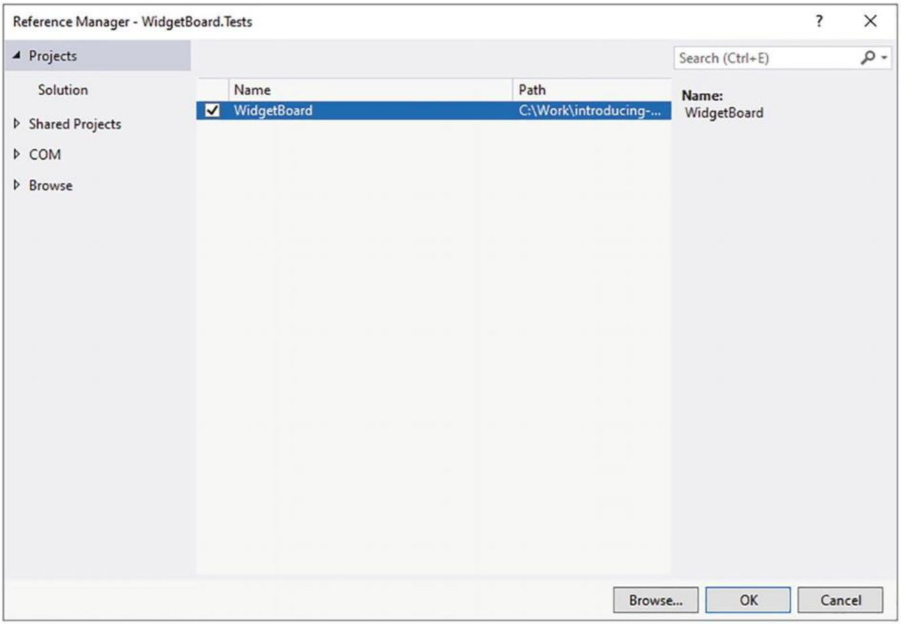

***Figure 13-2.** Reference Manager in Visual Studio*

5. Click **OK**.

Modify Your Test Project to Use MAUI Dependencies

The final step is to make your test project bring in the .NET MAUI dependencies just like the main application project. Open up the `WidgetBoard.Tests/WidgetBoard.Tests.csproj` file in Visual Studio Code or your favorite text editor and make the following changes.

The way to bring in the .NET MAUI dependencies is to add the NuGet packages to the test project. Normally you could just add this through Visual Studio or your favorite package manager UI, but we will do this manually and explain why afterward. Modify the `ItemGroup` element that contains `<PackageReference />` elements, which should now look like this; the changes are in **bold**:

```
<ItemGroup>
    <PackageReference Include="coverlet.collector"
    Version="6.0.2"/>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
    Version="17.11.1"/>
    <PackageReference Include="xunit" Version="2.9.2"/>
    <PackageReference Include="xunit.runner.visualstudio"
    Version="2.8.2"/>
    <PackageReference Include="Microsoft.Maui.Controls"
    Version="$(MauiVersion)"/>
    <PackageReference Include="Microsoft.Maui.Controls.
    Compatibility" Version="$(MauiVersion)"/>
</ItemGroup>
```

The reason why we have made this change manually is due to the `Version` value that we added; we did not add a specific version but instead used the variable `$(MauiVersion)`; this makes use of the installed version. Now you have set up everything ready to begin writing and running your unit tests.

Testing Your View Models

The MVVM architecture lends itself very well to unit testing each individual component.

First, you need to create a `ViewModels` folder in your *WidgetBoard.Tests* project and then add a new class file called `BoardDetailsPageViewModelTests.cs`. It is good practice to keep folders and tests named similarly to the code that they are testing to make it easier to organize and locate.

Now you can add in your first set of tests.

Testing BoardDetailsPageViewModel

Inside the class file that you just created, add the following:

```
[Fact]
public void SaveCommandCannotExecuteWithoutBoardName()
{
    var viewModel = new BoardDetailsPageViewModel(null, null);

    Assert.Equal(string.Empty, viewModel.BoardName);
    Assert.False(viewModel.SaveCommand.CanExecute(null));
}

[Fact]
public void SaveCommandCanExecuteWithBoardName()
{
    var viewModel = new BoardDetailsPageViewModel(null, null);

    viewModel.BoardName = "Work";
    Assert.True(viewModel.SaveCommand.CanExecute(null));
}
```

Testing INotifyPropertyChanged

I covered in Chapter 4 that `INotifyPropertyChanged` serves as the mechanism to keep your views and view models in sync; therefore, it can be really useful to verify that your view models are correctly implementing `INotifyPropertyChanged` by ensuring that it raises the `PropertyChanged` event when it should.

The following test shows how to create an instance of the `BoardDetailsPageViewModel`, subscribe to the `PropertyChanged` event, modify a property that you expect to fire the `PropertyChanged` event, and then `assert` that the event was invoked:

```
[Fact]
public void SettingBoardNameShouldRaisePropertyChanged()
{
    var invoked = false;
    var viewModel = new BoardDetailsPageViewModel(null, null);

    viewModel.PropertyChanged += (sender, e) =>
    {
        if (e.PropertyName?.Equals(nameof(BoardDetailsPage
        ViewModel.BoardName)) is  true)
        {
            invoked = true;
        }
    };
    viewModel.BoardName = "Work";
    Assert.True(invoked);
}
```

This provides you with the confidence to know that if the `BoardName` is not showing in your user interface, it will probably not be an issue inside the view model.

Testing Asynchronous Operations

Many modern applications involve some level of asynchronous operation, and a perfect example is your use of the Open Weather API in order to load the current location's weather. The `WeatherWidgetViewModel` relies on an implementation of the `IWeatherForecastService` interface you created in Chapter 11. Unit tests against specific implementations like this can be considered flaky. A flaky test is one that provides inconsistent results. Web service access can exhibit this type of behavior when unit testing given access limits on the API or other potential issues that could impact a reliable test run.

In order to remove test flakiness, you can create a mock implementation that will provide a set of consistent behavior.

Creating Your ILocationService Mock

Create a new folder in your *WidgetBoard.Tests* project and call it `Mocks`. I covered this before, but organizing your code in such a way really can make it much easier to maintain. With this new folder, you can create a new class file inside and call it `MockLocationService.cs`. Modify the contents to the following:

```
using WidgetBoard.Services;

namespace WidgetBoard.Tests.Mocks;

public class MockLocationService : ILocationService
{
    private readonly Location? location;
    private readonly TimeSpan delay;

    private MockLocationService(Location? mockLocation,
    TimeSpan delay)
    {
        location = mockLocation;
        this.delay = delay;
    }

    public static ILocationService ThatReturns(Location?
    location, TimeSpan after) =>
        new MockLocationService(location, after);

    public static ILocationService
    ThatReturnsNoLocation(TimeSpan after) =>
        new MockLocationService(null, after);
```

```
    public async Task<Location?> GetLocationAsync()
    {
        await Task.Delay(this.delay);
        return this.location;
    }
}
```

The implementation you provided for the `GetLocationAsync` method forces a delay based on the supplied `TimeSpan` parameter in the constructor to mimic a network delay and then return the location supplied in the constructor.

One key detail I really like to use when building mocks is to make the usage of them in my tests as easy to read as possible. You can see that the `MockLocationService` cannot be instantiated because it has a private constructor. This means that to use it, you must use the `ThatReturns` or `ThatReturnsNoLocation` method. Look at this and see how much more readable it is:

```
MockLocationService.ThatReturns(new Location(0.0, 0.0), after:
TimeSpan.FromSeconds(2));
```

The above is much more readable than the following because it includes the intent:

```
new MockLocationService(new Location(0.0, 0.0), TimeSpan.
FromSeconds(2));
```

Creating Your SecureStorage Mock

You can add a third file into the `Mocks` folder and call this class file `MockSecureStorage.cs`. Modify the contents to the following:

```
namespace WidgetBoard.Tests.Mocks;

public class MockSecureStorage : ISecureStorage
{
    private readonly Dictionary<string, string?> values = new();

    private MockSecureStorage(string key, string value)
    {
        values.Add(key, value);
    }

    public static MockSecureStorage ThatContains(string key,
    string value) =>
        new MockSecureStorage(key, value);

    public Task<string?> GetAsync(string key)
    {
        return Task.FromResult(values[key]);
    }

    public Task SetAsync(string key, string value)
    {
        values[key] = value;
        return Task.CompletedTask;
    }

    public bool Remove(string key)
    {
        if (values.ContainsKey(key))
        {
            values.Remove(key);
            return true;
        }

        return false;
    }
```

```
    public void RemoveAll()
    {
        values.Clear();
    }
}
```

The implementation you provided stores and retrieves the "secure" values inside a dictionary to mimic a real implementation on a device.

Creating Your WeatherForecastService Mock

You can add a third file into the Mocks folder and call this class file MockWeatherForecastService.cs. Modify the contents to the following:

```
using WidgetBoard.Communications;

namespace WidgetBoard.Tests.Mocks;

public class MockWeatherForecastService :
IWeatherForecastService
{
    private readonly Forecast? forecast;
    private readonly TimeSpan delay;

    private MockWeatherForecastService(Forecast? forecast,
    TimeSpan delay)
    {
        this.forecast = forecast;
        this.delay = delay;
    }

    public static IWeatherForecastService ThatReturns(Forecast?
    forecast, TimeSpan after) =>
        new MockWeatherForecastService(forecast, after);
```

```
    public static IWeatherForecastService
    ThatReturnsNoForecast(TimeSpan after) =>
        new MockWeatherForecastService(null, after);

    public async Task<Forecast?> GetForecast(double latitude,
    double longitude, string apiKey)
    {
        await Task.Delay(this.delay);
        return forecast;
    }
}
```

The implementation you provided for the `GetForecast` method forces a delay based on the supplied `TimeSpan` parameter in the constructor to mimic a network delay and then return the forecast supplied in the constructor.

Creating Your Asynchronous Tests

With your mocks in place, you can write tests that will verify the behavior of your application when calling asynchronous and potentially long running operations. You need to add a new class file to your `ViewModels` folder in the `WidgetBoard.Tests` project and call it `WeatherWidgetViewModelTests.cs` and then modify the contents to the following:

```
using WidgetBoard.Tests.Mocks;
using WidgetBoard.ViewModels;

namespace WidgetBoard.Tests.ViewModels;

public class WeatherWidgetViewModelTests
{
}
```

Now you can proceed to adding three tests to cover a variety of different scenarios.

```
[Fact]
public async Task NullLocationResultsInPermissionErrorState()
{
    var viewModel = new WeatherWidgetViewModel(
        MockWeatherForecastService.ThatReturnsNoForecast(after:
        TimeSpan.FromSeconds(5)),
        MockSecureStorage.ThatContains("OpenWeatherApiToken",
        "SomethingSecure"),
        MockLocationService.ThatReturnsNoLocation(after:
        TimeSpan.FromSeconds(2)));

    await viewModel.LoadWeatherForecast();

    Assert.Equal(State.PermissionError, viewModel.State);
    Assert.Equal(viewModel.Weather, string.Empty);
}
```

This first test, as the name implies, verifies that if a `null` location is returned from the `ILocationService` implementation, the view model `State` will be set to `PermissionError` and no `Weather` will be set.

```
[Fact]
public async Task NullForecastResultsInErrorState()
{
    var viewModel = new WeatherWidgetViewModel(
        MockWeatherForecastService.ThatReturnsNoForecast(after:
        TimeSpan.FromSeconds(5)),
        MockSecureStorage.ThatContains("OpenWeatherApiToken",
        "SomethingSecure"),
        MockLocationService.ThatReturns(new Location(0.0, 0.0),
        after: TimeSpan.FromSeconds(2)));
```

```
    await viewModel.LoadWeatherForecast();

    Assert.Equal(State.Error, viewModel.State);
    Assert.Equal(viewModel.Weather, string.Empty);
}
```

This second test, as the name implies, verifies that if a `null` forecast is returned from the `IWeatherForecastService` implementation, the view model `State` will be set to `Error` and no `Weather` will be set.

```
[Fact]
public async Task ValidForecastResultsInSuccessfulLoad()
{
    var weatherForecastService =
        MockWeatherForecastService.ThatReturns(
            new Communications.Forecast
            {
                Main = new Communications.Main
                {
                    Temperature = 18.0
                },
                Weather =
                [
                    new Communications.Weather
                    {
                        Icon = "abc.png",
                        Main = "Sunshine"
                    }
                ]
            },
            after: TimeSpan.FromSeconds(5));
```

```
    var locationService = MockLocationService.ThatReturns(
        new Location(0.0, 0.0),
        after: TimeSpan.FromSeconds(2));

    var viewModel = new WeatherWidgetViewModel(
        weatherForecastService,
        MockSecureStorage.ThatContains("OpenWeatherApiToken",
        "SomethingSecure"),
        locationService);

    await viewModel.LoadWeatherForecast();

    Assert.Equal(State.Loaded, viewModel.State);
    Assert.Equal("Sunshine", viewModel.Weather);
}
```

This final test, as the name implies, verifies that if a valid forecast is returned from the `IWeatherForecastService` implementation, the view model `State` will be set to `Loaded` and the `Weather` will be correctly set.

Testing Your Views

It is possible to write unit tests that will verify the behavior of your views.

Creating Your ClockWidgetViewModel Mock

In order to verify your `ClockWidgetView`, you need to provide it with a view model. Your `ClockWidgetViewModel` currently has some complexities in it that will make it difficult to use in the test. It displays the current date/time. Let's create a mock to remove this potential difficulty. Inside your `Mocks` folder, add a new class file called `MockClockWidgetViewModel.cs` and modify the contents to match the following:

```
using WidgetBoard.ViewModels;

namespace WidgetBoard.Tests.Mocks;

public class MockClockWidgetViewModel : IWidgetViewModel
{
    public int Position { get; set; }

    public string Type => "Mock";

    public MockClockWidgetViewModel(DateTime time)
    {
        Time = time;
    }

    public DateTime Time { get; }
    public Task InitializeAsync() => Task.CompletedTask;
}
```

Now you can use this in your unit tests to verify that your `ClockWidgetView` binds correctly to its view model.

Creating Your View Tests

First, create a `Views` folder in your *WidgetBoard.Tests* project and then add a new class file called `ClockWidgetViewTests.cs`.

```
using WidgetBoard.Tests.Mocks;
using WidgetBoard.Views;

namespace WidgetBoard.Tests.Views;

public class ClockWidgetViewTests
{
    [Fact]
    public void TextIsUpdatedByTimeProperty()
```

```
    {
        var time = new DateTime(2022, 01, 01);
        var clockWidget = new ClockWidgetView(null);
        Assert.Null(clockWidget.Text);

        clockWidget.WidgetViewModel = new MockClockWidget
        ViewModel(time);
        clockWidget.BindingContext = clockWidget.WidgetViewModel;

        Assert.Equal(time.ToString(), clockWidget.Text.Trim());
    }
}
```

The test `TextIsUpdatedByTimeProperty` creates a new `ClockWidgetView`, assigns your new `MockClockWidgetViewModel`, and then verifies that the Text property of the widget is correctly updated to reflect the value from the Time property on your view model through its binding.

Device Testing

Device testing is really a form of unit testing; however, it provides some unique abilities so it deserves its own top-level section. It essentially enables you to write unit tests that can be run on a device and therefore truly test any platform-specific pieces of functionality. A perfect example of this is to test the `PlatformLocationService` you implemented in the previous chapter to return the longitude and latitude coordinates of each platform provider's headquarters.

Creating a Device Test Project

You need to create another project in order to handle the running of the device tests. The documentation on the GitHub repository covers all that is needed, so go to `https://github.com/shinyorg/xunit-maui` but I will repeat the steps here for clarity.

Add a New Project to the Solution

- Right-click the “WidgetBoard” solution in Visual Studio.
- Select Add New Project....
- Select .NET MAUI Application.
- Enter the name WidgetBoard.DeviceTests.

Add the Device Test Runner NuGet Package

- Right-click the “WidgetBoard.DeviceTests” project in Visual Studio.
- Select Manage NuGet Packages....
- Enter *Shiny.Xunit.Runners.Maui* into the search bar.
- Select and install the package.

Remove Unnecessary Contents

The default template will create some entries that you don’t need, and I always recommend removing any bits of content that you don’t need. You can delete the following:

- MainPage.xaml and MainPage.xaml.cs
- App.xaml and App.xaml.cs
- AppShell.xaml and AppShell.xaml.cs
- All folders underneath /Resources except Styles

Replace CreateMauiApp

Inside the *MauiProgram.cs* file, the `CreateMauiApp` method can be replaced with the following:

```
public static MauiApp CreateMauiApp() => MauiApp
    .CreateBuilder()
    .ConfigureTests(new TestOptions
    {
        Assemblies =
        {
            typeof(MauiProgram).Assembly
        }
    })
    .UseVisualRunner()
    .Build();
```

The above code will register the application to rely on running a visual application; sadly there is not the ability to run a headless mode and therefore in an automated manner, but I am hopeful this will come in the future.

Add a Reference to the WidgetBoard Project

You can add the reference as follows:

- Right-click the WidgetBoard.DeviceTests project.
- Select Add ➤ Reference.
- Select Projects.
- Select WidgetBoard.
- Click Add.

This will result in *WidgetBoard.DeviceTests* depending on the *WidgetBoard* project file and will allow you to refer to the public contents of this project.

Remove Extra Entries in Project File

Modify the new WidgetBoard.DeviceTests.csproj project file by deleting the following section:

```
<ItemGroup>
    <MauiIcon Include="Resources\AppIcon\appicon.svg"
    ForegroundFile="Resources\AppIcon\appiconfg.svg"
    Color="#512BD4"/>

    <MauiSplashScreen Include="Resources\Splash\splash.svg"
    Color="#512BD4" BaseSize="128,128"/>

    <MauiImage Include="Resources\Images\*"/>
    <MauiImage Update="Resources\Images\dotnet_bot.png"
    Resize="True" BaseSize="300,185"/>

    <MauiFont Include="Resources\Fonts\*"/>

    <MauiAsset Include="Resources\Raw\**" LogicalName="%(
    RecursiveDir)%(Filename)%(Extension)"/>
</ItemGroup>
```

The reason for this is that by referencing the main app project, the build tasks detect duplicate files and will produce build errors. Note that if you are not referencing another .NET MAUI app project in your device runner test project, then you want to keep the above.

Adding a Device-Specific Test

Add a new folder and call it *Services*. Then add a new class file and call it *PlatformLocationServiceTests*.

In the new file, you can add the following contents:

```
using WidgetBoard.Services;
using Xunit;

namespace WidgetBoard.DeviceTests.Services;

public class PlatformLocationServiceTests
{
    [Fact]
    public async Task GetLocationAsyncWillReturnPlatform
    SpecificLocation()
    {
        var locationService = new PlatformLocationService();
        var location = await locationService.
        GetLocationAsync();

#if ANDROID
        Assert.Equal(37.419857, location.Latitude);
        Assert.Equal(-122.078827, location.Longitude);
#elif WINDOWS
        Assert.Equal(47.639722, location.Latitude);
        Assert.Equal(-122.128333, location.Longitude);
#else
        Assert.Equal(37.334722, location.Latitude);
        Assert.Equal(-122.008889, location.Longitude);
#endif
    }
}
```

The above is a relatively simple test, and in a real application, we would likely have concepts that are not hard-coded but this will prove the point.

Now that you have written your tests, you can run them on your devices.

Running Device-Specific Tests

In order to run your tests on a device, you first need to set your *WidgetBoard.DeviceTests* project as the startup project. You can do this as follows:

- Right-click the *WidgetBoard.DeviceTests* project in Solution Explorer.
- Select **Set as Startup Project**.

Now start the application from Visual Studio. Figure 13-3 shows the device test runner screen running on Windows.

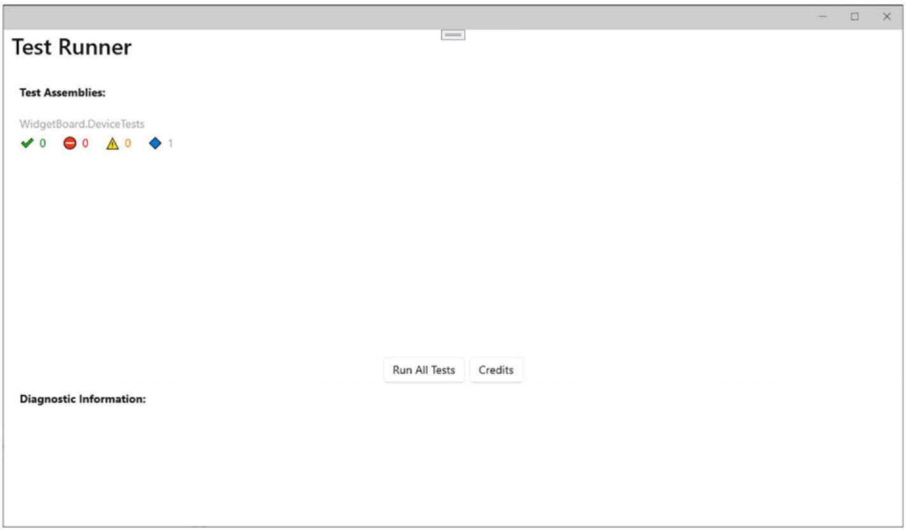

Figure 13-3. *Device test runner on the Windows platform*

You can click on a specific test and choose to run it, or you can simply **Run All Tests**. This part is entirely manual so it will require a human to perform these tasks but it can be left to run for as long as the tests need.

Finally, you will see the results of the test runs, and you can click them to see more information. Figure 13-4 shows the device test runner and a set of test results.

Figure 13-4. *Test run result for the GetLocationAsyncWillReturn PlatformSpecificLocation device test*

You can run these tests on all the platforms that you support to make sure that the code does what is expected.

Snapshot Testing

Snapshot testing is similar to unit testing, but it avoids the need to write `Assert` statements to manually define each expectation in the test. Instead, the result of a test is compared to a *golden master*. A golden master is a snapshot of a previous test run that you as the test author accept as the expected result for subsequent test runs. A snapshot can be anything ranging from a screenshot of the application to a serialization of an object in memory. If you take a look at the `WeatherWidgetViewModel` you unit tested in the earlier section, you can see that a serialization of the state of the `ValidForecastResultsInSuccessfulLoad` test will result in the following golden master being created:

```
{
  LoadWeatherCommand: {},
  IconUrl: https://openweathermap.org/img/wn/abc.png@2x.png,
  State: Loaded,
  Temperature: 18.0,
  Weather: Sunshine,
  Type: Weather
}
```

When this test is run, each time the serialized output of the `WeatherWidgetViewModel` will be compared to the above golden master. If any of the values are different from those in the golden master, the test will fail.

Snapshot Testing Your Application

In order to snapshot test your application, you will make use of the excellent library called *VerifyTests*. VerifyTests has some really great documentation and examples to get you started over at `https://github.com/VerifyTests/Verify`.

You will additionally need to consume the `Verify.Xunit` NuGet package. I have opted to create a separate project just to keep things clearly separated for the purpose of this example. You can repeat the steps in sections "Adding a Unit Test Project to Your Solution" and "Adding a Reference to the Project to Test," except that you will name the project *WidgetBoard.SnapshotTests*.

Add the Verify.Xunit NuGet Package

- Right-click the "WidgetBoard.SnapshotTests" project in Visual Studio.
- Select Manage NuGet Packages....
- Enter *Verify.Xunit* into the search bar.
- Select and install the package.

Add a Reference to the WidgetBoard and WidgetBoard.Tests Projects

You can add the reference as follows:

- Right-click the WidgetBoard.SnapshotTests project.
- Select Add ➤ Reference.
- Select Projects.
- Select WidgetBoard.
- Click Add.

Using `VerifyTests`, you can take a copy of your `WeatherWidgetViewModelTests` class in the *WidgetBoard.Tests* project and modify it to the following. The limited changes are shown in **bold** to highlight the differences from the original.

```
[UsesVerify]
public class WeatherWidgetViewModelTests
{
    [Fact]
    public async Task NullLocationResultsInPermission
    ErrorState()
```

```
{
    var viewModel = new WeatherWidgetViewModel(
    MockWeatherForecastService.ThatReturnsNoForecast(after:
    TimeSpan.FromSeconds(5)),
    MockSecureStorage.ThatContains("OpenWeatherApiToken",
    "SomethingSecure"),
    MockLocationService.ThatReturnsNoLocation(after:
    TimeSpan.FromSeconds(2)));

await viewModel.LoadWeatherForecast();

    await Verify(viewModel);
}

[Fact]
public async Task NullForecastResultsInErrorState()
{
    var viewModel = new WeatherWidgetViewModel(
    MockWeatherForecastService.ThatReturnsNoForecast(after:
    TimeSpan.FromSeconds(5)),
    MockSecureStorage.ThatContains("OpenWeatherApiToken",
    "SomethingSecure"),
    MockLocationService.ThatReturns(new Location(0.0, 0.0),
    after: TimeSpan.FromSeconds(2)));

    await viewModel.LoadWeatherForecast();

    await Verify(viewModel);
}

[Fact]
public async Task ValidForecastResultsInSuccessfulLoad()
{
    var weatherForecastService =
```

```
        MockWeatherForecastService.ThatReturns(
            new Communications.Forecast
            {
                Main = new Communications.Main
                {
                    Temperature = 18.0
                },
                Weather =
                [
                    new Communications.Weather
                    {
                        Icon = "abc.png",
                        Main = "Sunshine"
                    }
                ]
            },
            after: TimeSpan.FromSeconds(5));

    var locationService = MockLocationService.ThatReturns(
        new Location(0.0, 0.0),
        after: TimeSpan.FromSeconds(2));

    var viewModel = new WeatherWidgetViewModel(
        weatherForecastService,
        MockSecureStorage.ThatContains("OpenWeatherApiToken",
        "SomethingSecure"),
        locationService);

          await viewModel.LoadWeatherForecast();

        await Verify(viewModel);
    }
}
```

You remove the `Assert` statements and replace them by calling the `Verify` method. In your original scenario, you were only asserting a small number of things, but you can imagine that if the number of `Assert` statements were to grow, then this single method call to `Verify` really does reduce the complexity of your tests.

Brand-new tests will always fail until you accept the golden master. There is tooling that can make this task easier, which is again provided by the *VerifyTests* developers.

Passing Thoughts

I end this snapshot testing section with the statement that it is not for everyone. Some people really like the reduction in test case size, while it verifies more than most typical unit tests by the sheer fact that it verifies the whole object under test. As a counter argument, some people dislike that the expected state or golden master is in a file separate to the tests. I personally believe they provide great value, and I hope that this introduction to snapshot testing will give you enough context to decide whether it is going to be a good fit for you and your team, or at least give you the desire to experiment with the concept.

Summary

Now you have an overview of different testing techniques and the benefits they bring. You may prefer snapshot testing over writing your own asserts. I don't mind either way so long as you do test your code. We have not concluded all testing-related topics in this book; the next chapter covers automation testing.

In this chapter, you have

- Learned what testing is and why it is important
- Explored unit testing and how you can apply it to a .NET MAUI application
- Learned about snapshot testing and how you can implement it
- Explored what device tests are and how you can apply them to your applications

In the next chapter, you will

- Learn what automation testing is
- Gain insight into Appium, an automation testing framework
- Write automation tests that can interact with the widget board application you have been building

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch13`.

CHAPTER 14

Automation Testing

Abstract

The previous chapter covered a number of testing techniques to help prove how small units of code "do what they say on the tin." This chapter will broaden the scope of what is being tested into what would be considered integration or end-to-end testing. Furthermore, the focus of this chapter will be to focus on performing this testing in an automated manner – hence the title "Automation Testing."

What Is Automation Testing?

Automation testing is the process of using software to execute tests against an application in order to verify that the application both meets expectations and does so in a valid manner. One thing that a computer is excellent at is following instructions in a repeated manner; this makes automation testing a big advantage over manual testing due to the risk of a human making a mistake (I know I certainly do).

I must add I don't believe that automation testing will or should replace manual testing entirely, but it can serve as an excellent platform to save time and effort for the development/test team and allow them to focus on what needs to be added next. It also provides time for the team to perform more exploratory testing – this is the type of testing where you

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_14

don't necessarily follow a script of steps and instead just pretend to be a user or try to break things. Some great examples of exploratory testing in my past personal projects involve giving my wordsearch app to my then eight-year-old daughter and watch just how quickly she could do something entirely unexpected and break things.

Speaking from personal experience, being an indie developer makes it difficult enough to spend time building the application, let alone all of the testing effort required to ensure that the application is stable and does what I expect. I have found value in using automation test frameworks in the past to allow myself to modify the application and allow a computer to perform the testing that I have set up. In fact, it has caught some silly mistakes that I have made from late night coding sessions trying to get some new features shipped. I simply push the changes to the code repository and go to bed, all while a computer will compile those changes, deploy them, and then run my test suite against that new build. Great, right? Good then let's make this happen for our WidgetBoard application!

Automation Testing in .NET MAUI

In order to perform automation testing against our .NET MAUI application(s), we need to use a tool to perform the interaction with the application. The tool that we will be using in this book is called Appium. If you are not familiar with Appium, the following description from their website explains it well:

> "Appium is an open-source project and ecosystem of related software, designed to facilitate UI automation of many app platforms, including mobile (iOS, Android, Tizen), browser (Chrome, Firefox, Safari), desktop (macOS, Windows), TV (Roku, tvOS, Android TV, Samsung), and more!"

We have already covered that .NET MAUI supports multiple platforms, and thankfully all of the ones that it supports are also supported by Appium, making it a great candidate for automation testing.

Appium works by having a server execute the tests on a client application that runs on the target device. This means that running the tests will require an Appium server to run. Appium then provides drivers for each platform that you wish to test against. Let's proceed to following the steps to configure a testing environment to make this all clearer.

Installing Appium

The WidgetBoard application can run on Android, iOS, macOS, and Windows.

Installing Node.js

I mentioned earlier that Appium works by having a server execute the tests; for this reason, we need to install Node.js. In order to install the environment, you can follow the steps below:

1. Navigate to `https://nodejs.org/`.
2. Click Download Node.js.
3. Once downloaded, run the installer and follow through to completion.

Note for the purpose of the examples in this book, you will need to perform these steps on both your Windows and macOS machines. If you are only working on one operating system, then that is of course fine.

Install Appium

Now that you have Node.js installed, you should be able to open a terminal or command prompt session on either macOS or Windows and install Appium using node package manager (npm). Let's take a look at each in turn.

macOS

1. Open the **Terminal** application.
2. Enter the following command and then press return:

   ```
   sudo npm install -g appium
   ```

Windows

1. Open the **Command Prompt** application.
2. Enter the following command and then press return:

   ```
   npm install -g appium
   ```

This will find the package called `appium` via *node package manager*, and the `-g` argument will install it as a global tool. This means that you can just type `appium` into a future terminal or command prompt session and start an Appium server.

With Appium install, you can proceed to installing the relevant drivers for each platform that you wish to test.

Install Appium Driver for Android

The Android driver named *UIAutomator2* can only be installed on macOS or Windows, so open the Terminal or Command Prompt application, enter the following command, and then press return:

```
appium driver install uiautomator2
```

The installation should complete and report the following:

```
✔ Checking if 'appium-uiautomator2-driver' is compatible
✔ Installing 'uiautomator2' using NPM install spec 'appium-
uiautomator2-driver'
i Driver uiautomator2@3.8.1 successfully installed
- automationName: UiAutomator2
- platformNames: ["Android"]
```

Install Appium Driver for iOS

The iOS driver named *XCUITest* can only be installed on macOS, so open the Terminal application, enter the following command, and then press return:

```
appium driver install xcuitest
```

The installation should complete and report the following:

```
✔ Checking if 'appium-xcuitest-driver' is compatible
✔ Installing 'xcuitest' using NPM install spec 'appium-
xcuitest-driver'
i Driver xcuitest@7.28.3 successfully installed
- automationName: XCUITest
- platformNames: ["iOS","tvOS"]
```

Install Appium Driver for macOS

The macOS driver named *Mac2* can only be installed on macOS, so open the Terminal application, enter the following command, and then press return:

```
appium driver install mac2
```

The installation should complete and report the following:

```
✔ Checking if 'appium-mac2-driver' is compatible
✔ Installing 'mac2' using NPM install spec 'appium-
mac2-driver'
i Driver mac2@1.20.2 successfully installed
- automationName: Mac2
- platformNames: ["Mac"]
```

Install Appium Driver for Windows

The Windows driver named *Windows* can only be installed on Windows, so open the Command Prompt application, enter the following command, and then press return:

```
appium driver install windows
```

The installation should complete and report the following:

```
✔ Checking if 'appium-windows-driver' is compatible
✔ Installing 'windows' using NPM install spec 'appium-
windows-driver'
i Driver windows@3.0.2 successfully installed
- automationName: Windows
- platformNames: ["Windows"]
```

Install and Run WinAppDriver

An additional prerequisite is required to run Appium on Windows, and that is to install WinAppDriver. The project repository is available at `https://github.com/microsoft/WinAppDriver`, and the steps to follow are as follows:

1. Download Windows Application Driver installer version 1.2.1 from https://github.com/Microsoft/WinAppDriver/releases.

2. Run the installer on a Windows 10 or 11 machine where your application under test is installed and will be tested.

3. Enable Developer Mode in Windows settings.

4. Run WinAppDriver.exe from the installation directory (e.g., C:\Program Files (x86)\Windows Application Driver).

Windows Application Driver will then be running on the test machine listening to requests on the default IP address and port (127.0.0.1:4723). This will be fine for the purpose of our testing, but you do have the ability to configure the address and port should you wish.

This concludes the driver installation; now let's take a quick look at a very useful tool and then move onto writing some tests.

Install Appium Inspector

This tool is a must! I owe a big shout-out to my good friend and the reviewer of this book, Gerald, for introducing me to this tool when I was facing some pain trying to correctly identify parts of the application to test. Figure 14-1 shows the Appium inspector and how it can be configured to start an iOS session.

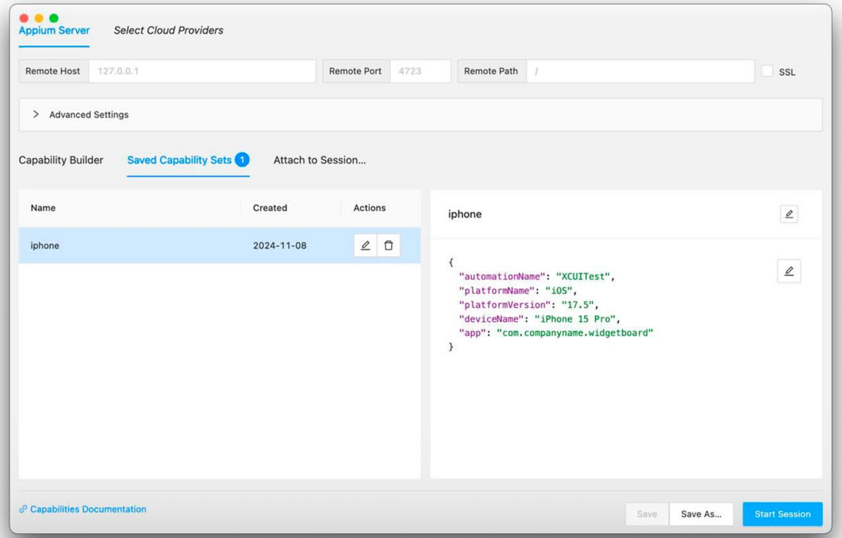

Figure 14-1. *The Appium inspector application, configured to connect to the iOS version of the application*

The inspector application makes it possible to interact and inspect the application being targeted. Figure 14-2 shows the inspector with the Add button selected from the `BoardListPage` and the properties available for inspection and verification in our tests.

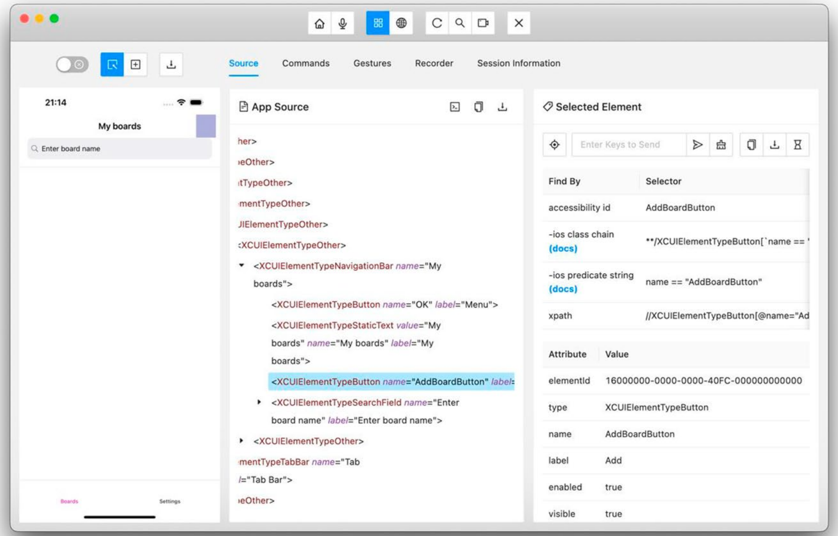

Figure 14-2. *The Appium inspector application showing available information for the Add button*

This concludes the steps on how to install Appium and all of its dependencies, so let's now proceed to creating our test project and tests.

Creating the Automation Test Project

One key detail to highlight is that Appium does not provide the mechanism to execute tests; it purely acts as the mechanism to interact with an application. In order to write and execute tests, we will need to make use of a unit testing framework. We made use of xUnit in the previous chapter, and while I would strongly recommend consistency when using frameworks like this, I wanted to give you exposure to a second test framework to help you decide which might be the right framework for you and your team. In this chapter, we will be making use of NUnit. Let's proceed to creating the project and then make use of Appium.

You will need to add a new project to the WidgetBoard solution. Most public samples at the time of writing make use of a large number of projects – one shared project and then one project per platform under test. I have opted for a single project with the aim of reducing the complexity while providing more flexibility in terms of how to drive tests on not just multiple platforms but also multiple devices on each platform. Let's create the project and then work through how it will all fit together.

1. Click the **File** menu.
2. Click **Add**.
3. Click **New Project**.
4. Enter *Test* in the **Search for templates** box.
5. Select **NUnit Test Project**.
6. Click **Next**.
7. Enter a name for the project. I opted for *WidgetBoard.AutomationTests*.
8. Click **Next**.
9. Select the framework. The default should be fine; just make sure it matches the target version of the .NET MAUI application project.
10. Click **Create**.

One key detail to note is that this new project does not target each of the platforms (e.g., net9.0-ios); it just targets plain .NET (net9.0). This is due to the fact that it is "just" a test project; you can and will make use of the `dotnet test` command-line option to run these tests. What will happen is during the execution of a test, the code will interact with an application running on the target under test (e.g., iOS) and verify that it behaves as expected.

I mentioned that this book is doing things a little bit differently to most public samples. I don't want to take away from the work of these samples. If you are happy to follow this approach, then I should highlight that my good friend Gerald and the reviewer of this book has created a template to make the steps that you just followed a lot simpler. You can check the details out for this at `https://github.com/jfversluis/Template.Maui.UITesting`.

Now that you have created the project, you will proceed to introduce some helper implementations to initialize the Appium layer so that the tests can interact with the application on each target platform.

Add the Appium NuGet Package

You can do this by following these steps:

- Right-click the WidgetBoard project.
- Click **Manage NuGet Packages**.
- In the Search field, enter *Appium.WebDriver*.
- Select the **Appium.WebDriver** package and select **Add Package**.
- A confirmation dialog will show. Review and accept the license details if you are happy.

Creating an Appium Server

I explained earlier that Appium uses a server to execute the interaction with the application under test; therefore, you will need to add some code to start an Appium server instance on your machine. In order to achieve this, you can create an instance of the `AppiumLocalService`, configure it to the address and port the server is running on, and then start it. Let's create a new class file, call it AppiumServerHelper, and then modify its contents to the following:

```
using OpenQA.Selenium.Appium.Service;

namespace WidgetBoard.AutomationTests;

public static class AppiumServerHelper
{
    private static AppiumLocalService? appiumLocalService;

    private const string DefaultHostAddress = "127.0.0.1";
    private const int DefaultHostPort = 4723;

    public static void StartAppiumLocalServer(
        string host = DefaultHostAddress,
        int port = DefaultHostPort)
    {
        if (appiumLocalService is not null)
        {
            return;
        }

        var builder = new AppiumServiceBuilder()
            .WithIPAddress(host)
            .UsingPort(port);

        // Start the server with the builder
        appiumLocalService = builder.Build();
        appiumLocalService.Start();
    }
```

```
    public static void DisposeAppiumLocalServer()
    {
        appiumLocalService?.Dispose();
    }
}
```

The code that you just added will build a service to connect to a default address of `127.0.0.1` and port of `4723`. For the examples in this book, the default values are fine, but if you opt for using different values in your Appium server, then remember to update these.

Now that you have the code to start the server, the next step is to create a driver instance for the platform being tested. Let's proceed to doing this now.

Creating the Appium Platform Drivers

The first step of this section will be to create the bootstrapping code for the tests that will be run; then you will add each platform-specific driver implementation separately. The bootstrapping and teardown will make use of the `SetUpFixture`, `OneTimeSetup,` and `OneTimeTearDown` attributes that are provided by NUnit. These enable you to define a method that will be executed once per test assembly before the tests begin and also one when the tests finish. This is perfect for the scenario of starting the Appium server and then stopping it respectively.

Let's add a new class file, call it *AppiumSetup*, and then replace the contents with the following:

```
using NUnit.Framework;
using OpenQA.Selenium.Appium;
using OpenQA.Selenium.Appium.Android;
using OpenQA.Selenium.Appium.Enums;
using OpenQA.Selenium.Appium.iOS;
```

```
using OpenQA.Selenium.Appium.Mac;
using OpenQA.Selenium.Appium.Windows;

namespace WidgetBoard.AutomationTests;

[SetUpFixture]
public class AppiumSetup
{
    private static AppiumDriver? driver;

    public static AppiumDriver App => driver ?? throw new NullR
    eferenceException("AppiumDriver is null");

    [OneTimeSetUp]
    public void RunBeforeAnyTests()
    {
        AppiumServerHelper.StartAppiumLocalServer();

        driver = CreateDriver();
    }

    private static AppiumDriver CreateDriver()
    {
        var platformName = TestContext.
        Parameters["platformName"];

        switch (platformName)
        {
            case "Android":

            case "iOS":

            case "Mac":

            case "Windows":
        }
```

```
        return null;
    }

    [OneTimeTearDown]
    public void RunAfterAllTests()
    {
        driver?.Quit();

        // If an Appium server was started locally above, make
        sure we clean it up here
        AppiumServerHelper.DisposeAppiumLocalServer();
    }
}
```

The `CreateDriver` method makes use of the `TestContext.Parameters` property; this makes it possible to parameterize a full suite of tests; this is especially useful when wanting to repeat the same tests on multiple different devices for the same platform. Trust me when I say that the vast number of Android manufacturers and devices means that being able to do this in an automated manner can really help to pinpoint where issues might lie! The values in `TestContext.Parameters` are populated through a .runsettings file that is passed to the test runner when the unit tests are executed. As you build the driver code, you will see more use of the `TestContext.Parameters` property; once the driver code has been introduced to the code base, we will then look to create each of the .runsettings files for each platform.

The `RunBeforeAnyTests` and `RunAfterAllTests` methods make use of the `AppiumServerHelper` class that you introduced in the previous section. They also make use of the `driver` field which should be populated from the `CreateDriver` method; this leads us nicely onto creating each platform-specific driver.

Creating the Android Driver

The Android driver can take a number of different values based on whether you are testing a release or a debug build of the application; this book focuses on testing a release build as that is a more likely scenario, but the code in the sample repository for this book also includes the required code to test against a debug build. Inside the `switch` statement in the `CreateDriver` method, we can add the following changes inside the `case "Android":` section:

```
case "Android":
    var androidOptions = new AppiumOptions
    {
        AutomationName = "UIAutomator2",
        PlatformName = platformName,
        App = TestContext.Parameters["app"]
    };

    return new AndroidDriver(androidOptions);
```

You may have wondered why I included the console results for each platform in order to show that each driver had been installed successfully; if you look back now, you should notice that the `PlatformName` and `AutomationName` entries match those reported in the console results.

Note if you wish to run your tests on an Android Virtual Device (AVD) or emulator, then you need to supply an additional option *avd*. This option can be added via the AddAdditionalAppiumOption called avd; if you supply this, it will boot an Android emulator up that matches the name supplied, for example:

```
androidOptions.AddAdditionalAppiumOption("avd", "pixel_5_-_
api_33");
```

You did not supply this in your code, so in order to test your application, you must make sure that the Android emulator is booted manually.

Creating the iOS Driver

The iOS driver initialization is the main reason why I have opted to make use of the "*runsettings*" configuration file for unit tests. You haven't created any runsettings files; don't worry, these will follow along after creating the code to instantiate the platform drivers.

Inside the `switch` statement in the `CreateDriver` method, we can add the following changes inside the `case "iOS":` section:

```
case "iOS":
    var iOSOptions = new AppiumOptions
    {
        AutomationName = "XCUITest",
        PlatformName = platformName,
        PlatformVersion = TestContext.
        Parameters["platformVersion"],
        DeviceName = TestContext.Parameters["deviceName"]
        App = TestContext.Parameters["app"]
    };

    return new IOSDriver(iOSOptions);
```

Note that the value for App can be the full path to the .app file to test or the bundle ID if the app is already installed on the device.

Creating the macOS Driver

Inside the `switch` statement in the `CreateDriver` method, we can add the following changes inside the `case "Mac":` section:

```
case "Mac":
    var macOSOptions = new AppiumOptions
    {
        AutomationName = "mac2",
        PlatformName = platformName,
        App = TestContext.Parameters["app"]
    };

    return new MacDriver(macOSOptions);
```

Note that the value for `App` can be the full path to the .app file to test or the bundle ID if the app is already installed on the device.

Creating the Windows Driver

Inside the `switch` statement in the `CreateDriver` method, we can add the following changes inside the `case "Windows":` section:

```
case "Windows":
    var windowsOptions = new AppiumOptions
    {
        AutomationName = "windows",
        PlatformName = platformName,
        App = TestContext.Parameters["app"]
    };

    return new WindowsDriver(windowsOptions);
```

Note that the value for `App` needs to be the identifier of the deployed application. Therefore, the application must be deployed before testing.

This concludes the changes required to instantiate an Appium driver on each of the target platforms; let's proceed to looking at the configuration files required for each platform and then actually write some tests.

Parameterizing the Tests

I have explained why we are parameterizing the tests. I should make it clear that this level of parameterizing tests is for the whole suite – basically all tests in this assembly. We will look at a further example of how to parameterize an individual test to make it possible to keep the amount of code required down to a minimum. The parameters for the test suite are through a .runsettings file; this is an XML file that can contain a set of parameters that a test can access.

Configuration for Android

The Android configuration can be added by adding a new file to the project. Call the file *android.runsettings* and then modify the contents to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <TestRunParameters>
        <Parameter name="platformName" value="Android" />
        <Parameter name="app" value="<OUTPUT_DIRECTORY_PATH>
        /com.companyname.widgetboard-Signed.apk" />
    </TestRunParameters>
</RunSettings>
```

Configuration for iOS

The iOS configuration can be added by adding a new file to the project. Call the file *iphone-15pro-17.5.runsettings* and then modify the contents to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <TestRunParameters>
        <Parameter name="platformName" value="iOS" />
        <Parameter name="platformVersion" value="17.5" />
        <Parameter name="deviceName" value="iPhone 15 Pro" />
        <Parameter name="app" value="com.companyname.
        widgetboard" />
    </TestRunParameters>
</RunSettings>
```

Hopefully the file name alone will start to give the impression of how we can customize the test run; we could easily introduce a second .runsettings file to support testing the application running on an iPad Pro 13 inch or another iPhone variant. This means that all of the same tests can be executed against different devices without any code changes.

Configuration for macOS

The macOS configuration can be added by adding a new file to the project. Call the file *macos.runsettings* and then modify the contents to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <TestRunParameters>
        <Parameter name="platformName" value="Mac" />
        <Parameter name="app" value="com.companyname.
        widgetboard" />
    </TestRunParameters>
</RunSettings>
```

Configuration for Windows

The Windows configuration can be added by adding a new file to the project. Call the file *windows.runsettings* and then modify the contents to match the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
    <TestRunParameters>
        <Parameter name="platformName" value="Windows" />
        <Parameter name="app" value="com.companyname.
        widgetboard_9zz4h110yvjzm!App" />
    </TestRunParameters>
</RunSettings>
```

Configuring Visual Studio to Use a runsettings File

In Visual Studio, it is possible to select a runsettings file that will be used for test executions. You can follow these steps below:

- Select **Test**.
- Select **Configure Run Settings**.
- Select **Select Solution Wide runsettings File**, and then select a .runsettings file that you created above.

This will then result in test runs within Visual Studio using that file.

Running Tests from Command Line/Terminal

There currently aren't any tests that exist in our test project but given we have discussed the .runsettings file we can run the following command

```
dotnet test WidgetBoard.AutomationTests --settings
iphone-15pro-17.5.runsettings
```

The above command makes it possible to include this in your CI/CD processes in order to automatically execute the automation tests.

Writing the Automation Tests

This chapter has most likely felt pretty heavy up until this point without having anything to really show for it. I do apologize for that, but sadly there is a fair amount of setup required to first get our environment ready and also some prerequisite information required for the approach taken. This section will now provide concrete examples of how to interact with a .NET MAUI application and use those interactions to verify the application behaves as expected.

The tests that will be created in this section will be against the `BoardListPage`, `BoardDetailsPage,` and `FixedBoardPage` implementations. This detail is especially important because we need to make sure that it is possible to interact with the controls on those pages. In order to make this task possible, you will want to make use of the `AutomationId` property that exists on .NET MAUI controls.

The `AutomationId` property provides developers with a mechanism to expose a unique identifier on controls to make it possible to uniquely identify them in the visual tree when automating UI. There will be times when an `AutomationId` is not possible, for example, if you are using a third-party control that hasn't added `AutomationIds` into their implementation. Don't worry, the final test that will be created will show how to use XPaths to work around this.

Let's proceed to creating a new class file in the *WidgetBoard.AutomationTests* project, call it *BoardTests.cs*, and modify it to have the following contents:

```
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Appium;
```

```
using OpenQA.Selenium.Appium.Windows;

namespace WidgetBoard.AutomationTests;

public class BoardTests
{
    private AppiumDriver App => AppiumSetup.App;

    // This could also be an extension method to AppiumDriver
    if you prefer
    private AppiumElement FindUIElement(string id)
    {
        if (App is WindowsDriver)
        {
            return App.FindElement(MobileBy.
            AccessibilityId(id));
        }

        return App.FindElement(MobileBy.Id(id));
    }
}
```

The code added makes use of the prerequisite driver implementation that you added a short while ago and also introduces the `FindUIElement` method; this method is purely to deal with how Windows behaves slightly differently to the other platforms.

Testing the Add New Board Button

The entry point to the application is to show a blank list of boards; at the top of the screen is a button that will allow the user to create a new board. Inside the *BoardTests.cs* file, add the following method:

```
[Test, Order(1)]
public void SaveButtonIsDisabledByDefault()
{
    FindUIElement("AddBoardButton").Click();

    var saveButton = FindUIElement("SaveButton");

    Assert.That(saveButton.Enabled, Is.False);

    FindUIElement("Cancel").Click();
}
```

This method doesn't just test that the add button works, it does a number of other things; let's break it down line by line to understand it in more detail.

```
[Test, Order(1)]
```

The above states that the method is a test in the NUnit framework and that it will be the first test to be executed.

```
    FindUIElement("AddBoardButton").Click();
```

We will make use of the new method we added a short while ago; it will attempt to find a UI element with the `AutomationId` of `"AddBoardButton"` and then interact with it. I use the word "interact" because on a mobile platform, there isn't really a click interaction but a touch interaction. If Appium cannot find the UI element, an exception will be thrown stating it couldn't find it; this works as an implicit assertion because the test will fail and it will fail with a useful error message.

The result of the `Click` method call will result in the `BoardDetailsPage` being presented to the user in the application.

```
    var saveButton = FindUIElement("SaveButton");
```

The above will attempt to find a UI element with the `AutomationId` of `"SaveButton"`.

```
Assert.That(saveButton.Enabled, Is.False);
```

The above line will verify that the save button is currently disabled; this is down to the application requiring a name to be provided for the board. This test would then catch a regression if a developer accidentally turned off that rule.

```
FindUIElement("Cancel").Click();
```

This final step is not really part of the test, but it resets the state of the application by navigating back to the `BoardListPage`. Some test enthusiasts might highlight how tests should not rely on or affect the outcome of other tests, and in the majority of scenarios, I would agree. In this scenario, however, I believe there is value in testing each individual unit of functionality but also testing them in combination to prove that the units integrate with each other. Plus sometimes you need data in your application to test specific scenarios and what better way to create the data than through automation?

The test made use of AutomationIds which the application code does not currently support; let's add in the ones to support this test; open the *BoardListPage.xaml* file and modify the ToolbarItem to match the following (changes in **bold**):

```
<ContentPage.ToolbarItems>
    <ToolbarItem
        Text="Add"
        AutomationId="AddBoardButton"
        Command="{Binding AddBoardCommand}" />
</ContentPage.ToolbarItems>
```

You should notice how the `AutomationId` you just added matches that used in the test. You will additionally need to modify the *BoardDetailsPage.xaml* file to introduce two further `AutomationIds`; make the following changes in **bold**:

```
<Grid ColumnDefinitions="*,*,*">
    <Button
        Text="Cancel"
        Command="{Binding CancelCommand}"
        AutomationId="CancelButton" />

    <Button
        Text="Save"
        Grid.Column="2"
        Command="{Binding SaveCommand}"
        AutomationId="SaveButton" />
</Grid>
```

This makes it possible to assert that the save button is disabled and then to interact with the cancel button.

Adding a Test to Create Boards

At times you might want to execute the steps in a test multiple times to work through different scenarios or in our case create multiple sets of data. It is extremely valuable to run these as separate tests because it means if one scenario fails, it won't impact another scenario that follows, and we can do this without having to copy and paste tests around. The `TestCase` attribute provided by NUnit allows you to define data values that will be passed into a test as parameters, and the test will be executed for each `TestCase` attribute added. Let's see this in action by creating a test that will execute two times and result in creating two differently named boards. You can add the following code into the *BoardTests.cs* file:

```
[Order(2)]
[TestCase("Work", 4, 4)]
[TestCase("Family", 2, 2)]
public void CanSaveBoard(string boardName, int numberOfColumns,
int numberOfRows)
{
    FindUIElement("AddBoardButton").Click();

    FindUIElement("BoardNameEntry").SendKeys(boardName);
    FindUIElement("NumberOfColumnsEntry").
    SendKeys(numberOfColumns.ToString());
    FindUIElement("NumberOfRowsEntry").SendKeys(numberOfRows.
    ToString());

    var saveButton = FindUIElement("SaveButton");

    Assert.That(saveButton.Enabled, Is.True);
    saveButton.Click();

    var createdBoard = FindUIElement(boardName);

    Assert.That(createdBoard, Is.Not.Null);
    Assert.That(createdBoard.Displayed, Is.True);
}
```

I won't break down each line this time as a fair amount should feel familiar; I will highlight the new concepts though. As I mentioned, this test will be executed twice: once with a `boardName` of `"Work"` and once with a `boardName` of `"Family"`.

```
FindUIElement("BoardNameEntry").SendKeys(boardName);
```

This will result in the value inside the `boardName` parameter being entered into the entry field with `AutomationId` of `BoardEntryName`.

```
    var createdBoard = FindUIElement(boardName);

    Assert.That(createdBoard, Is.Not.Null);
    Assert.That(createdBoard.Displayed, Is.True);
```

The three lines above are called after the save happens; this is important to highlight because it means that the application will have navigated back to the `BoardListPage`. These lines above will find the `Label` inside the `CollectionView` with `AutomationId` of the board name and then verify that it is visible to the user through the `Displayed` property. This is something I wanted to highlight because the properties available in the Appium layer are designed to be platform agnostic and therefore do not directly match the .NET MAUI property names.

Adding a Test to Interact with a CollectionView

The final test that you will be adding in this chapter shows how to make use of the XPath lookup rather than by `AutomationId`; it also shows how you can select an item in a `CollectionView` once found in order to mimic the user selecting a board to view. You can add the following code to the *BoardTests.cs* file:

```
[Test, Order(3)]
public void CanSelectEntryInListOfBoards()
{
App.FindElement(By.XPath("//XCUIElementTypeStaticText[
@name='Work']")).Click();

    var grid = FindUIElement("BoardGrid");

    Assert.That(grid, Is.Not.Null);
    Assert.That(grid.Displayed, Is.True);
}
```

The only new concept is the use of the `By.XPath` method; this is a helper method provided by Appium to perform a lookup by XPath. I won't be digging any further into XPath or the complicated scenarios you can build using it, but hopefully the above example shows how you can build a relatively straightforward lookup. The code above looks for an element of type `XCUIElementTypeStaticText` which has the name of `'Work'`. It is worth noting that lookups by XPath typically perform worse than using the `AutomationId`; in our scenario, we could use the `AutomationId`, but this example was just to show how you could build it another way.

Now that you have some insight into how you can instruct Appium to discover and select elements within your application's visual tree, I would like to highlight another library that can help to simplify the code that you need to write. This library is available at `https://github.com/jfversluis/Plugin.Maui.UITestHelpers`.

This now concludes the chapter on automation testing and testing in general in the book. If you were to run the tests inside your IDE or using the command-line tooling, you should be able to observe the application being tested under automation.

Summary

Now you have an overview of automation testing, the required effort to set up a system to support it through Appium, and some techniques to use when writing automation tests. I really hope this chapter has been as enjoyable learning about the options as it has been to expose them. It really is impressive what you can automate through a framework like Appium!

In this chapter, you

- Learned what automation testing is
- Gained insight into Appium, an automation testing framework
- Wrote automation tests that can interact with the widget board application you have been building

In the next chapter, you will

- Learn what .NET MAUI Graphics is
- Gain an insight into some of the power provided by .NET MAUI Graphics
- Build your own sketch widget with the .NET MAUI `GraphicsView` control
- Further enhance the WidgetBoard application by adding a graphical element to the clock widget

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch14`.

CHAPTER 15

Let's Get Graphical

Abstract

In this chapter, you will learn what .NET MAUI Graphics is, how it can be used, and some practical examples of why you would want to use it. You will also gain insight into some of the power provided by .NET MAUI Graphics and how you can use it to build your own sketch widget with the .NET MAUI `GraphicsView` control.

.NET MAUI Graphics

.NET MAUI Graphics is another one of my favorite topics! I am currently exploring the idea of building a game engine on top of it given the amount of power it already offers. If you are interested in the game engine, please feel free to check out the repository on GitHub at `https://github.com/bijington/orbit`.

It has the potential to offer the ability for so much to be achieved, things like rendering chart controls or other fancy concepts all through a cross-platform API, meaning you only really need to focus on the problems you are trying to solve and not worry about each individual platform.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_15

Essentially .NET MAUI Graphics offers a surface that can render pixel-perfect graphics on any platform supported by .NET MAUI. Consider .NET MAUI Graphics as an abstraction layer, like .NET MAUI itself, on top of the platform-specific drawing libraries. So we get all the power of each platform but with a simple unified .NET API that we as developers can work with.

Drawing on the Screen

.NET MAUI provides `GraphicsView`, which you can use to draw shapes on the screen. You need to assign the `Drawable` property on `GraphicsView` with an implementation that knows how to draw. This implementation must implement the `IDrawable` interface that defines a `Draw` method.

Updating the Surface

In order to trigger the application or `GraphicsView` to update what is rendered on screen, you must call the `Invalidate` method on `GraphicsView`. This will then cause the `IDrawable.Draw` method to be invoked, and your code will be given the chance to update the canvas.

The way to interact with the `ICanvas` implementation is to first set the values you need such as fill color (`FillColor`) or stroke color (`StrokeColor`) and then call the draw method you are interested in (`FillSquare()` or `DrawSquare()`, respectively).

Let's look at some basic examples to get a better understanding of how to use the graphics layer.

Drawing a Line

Inside the `Draw` method, you can interact with the `ICanvas` to draw a line using the `DrawLine` method. The following code shows how this can be achieved:

```
public void Draw(ICanvas canvas, RectF dirtyRect)
{
    canvas.StrokeColor = Colors.Red;
    canvas.StrokeSize = 6;
    canvas.DrawLine(0, 20, 100, 50);
}
```

You set `StrokeColor` and `StrokeSize` before calling the `DrawLine` method. Order is important, and you must set these properties before you draw. Figure 15-1 shows the result of the `Draw` method from above.

Figure 15-1. *Drawing a line in .NET MAUI Graphics*

In addition to drawing lines, you can draw many different shapes such as ellipse, rectangle, rounded rectangle, and arc. You can draw even more complex shapes through paths.

Drawing a Path

Paths are not to be confused with the Shapes API provided with .NET MAUI. Paths in .NET MAUI Graphics enable you to build up a set of coordinates in order to draw a more complex shape.

```
public void Draw(ICanvas canvas, RectF dirtyRect)
{
    PathF path = new PathF();
    path.MoveTo(40, 10);
    path.LineTo(70, 80);
    path.LineTo(10, 50);
    path.Close();
```

```
    canvas.StrokeColor = Colors.Red;
    canvas.StrokeSize = 6;
    canvas.DrawPath(path);
}
```

You first build up a PathF through the MoveTo, LineTo, and Close methods. The MoveTo method moves the current location of the path to the specified coordinates, and then the LineTo method draws a line from the current location that you just set in MoveTo to the coordinates specified in the LineTo method call. Finally, the Close method allows you to close the path. This means that the final location will have a line added back to the starting location. Notice that you didn't explicitly add a LineTo(40, 10) method call in; Close does this for you. Then you set the StrokeColor and StrokeSize before calling the DrawPath method. Figure 15-2 shows the result of the Draw method from above.

Figure 15-2. *Drawing a path in .NET MAUI Graphics*

It is this DrawPath method that you will be utilizing in the new widget you will be building as part of this chapter.

Maintaining the State of the Canvas

There can be times when you want to preserve some of the settings that you apply to the canvas, such as properties like StrokeColor and FillColor. All properties related to Stroke and Fill, plus others like transformation properties, can be preserved. This can be done through the SaveState method, which will save the current state. This saved state can then be restored through the RestoreState method. It is also possible to reset the current graphics state back to the default values with the `ResetState` method. These three methods can provide a large amount of functionality in specific scenarios. Say you have implemented a chart rendering control where the chart is rendered and then each individual series is rendered separately. You want to preserve the state of the chart's graphics settings but wish to reset each time you render a series (e.g., each column in a bar chart).

Further Reading

You have only scratched the surface of what is possible with the .NET MAUI Graphics layer. I strongly recommend that you refer to the Microsoft documentation at https://learn.microsoft.com/dotnet/maui/user-interface/graphics/ where it shows much more complex scenarios such as painting patterns, gradients, images, rendering text, and much more.

Building a Sketch Widget

My daughters love to doodle and leave me little notes when I am away from my desk, so I thought why not give them the ability to draw digital sketches and help save some trees. Let's create a new widget and then piece together this new drawing mechanic.

Creating the SketchWidgetViewModel

As with all of the widgets, you want to create a view model to accompany the view. Let's add a new class file into the `ViewModels` folder and call it `SketchWidgetViewModel.cs`. Modify it with the following contents:

```
namespace WidgetBoard.ViewModels;

public class SketchWidgetViewModel : IWidgetViewModel
{
    public const string DisplayName = "Sketch";
    public int Position { get; set; }
    public string Type => DisplayName;
}
```

The view model is relatively simple as it only really needs to implement the basics of the `IWidgetViewModel` interface. If you decide to add more functionality into your widget, you have the infrastructure in place to do so.

Let's now deal with the view and user interaction.

Representing a User Interaction

When a user interacts with the new widget, they will be drawing on the screen. You will need to record this interaction so that it can be rendered inside the `Draw` method that the `SketchWidgetView` implements through the `IDrawable` interface. Add a new class file, call it `DrawingPath.cs` in the root of the project, and modify it to have the following contents:

```
public class DrawingPath
{
    public DrawingPath(Color color, float thickness)
    {
        Color = color;
        Thickness = thickness;
        Path = new PathF();
    }
    public Color Color { get; }
```

```
    public PathF Path { get; }
    public float Thickness { get; }
    public void Add(PointF point) => Path.LineTo(point);
}
```

The class has three main properties:

- `Color` represents the color of the line being drawn.
- `Thickness` represents how thick the line is.
- `Path` contains the points that make up the line.

You also have a single method that adds a new point into the `Path` property. This ties in well with the .NET MAUI Graphics layer as you receive the point when the user interacts with the surface and then you can also use the same type to render the line on the screen.

Let's create the widget view that will make use of this class.

Creating the SketchWidgetView

As with each of the widget views, you will be creating a XAML-based view. It will be inside the view where most of the logic resides because this widget is largely view related.

Add a new .NET MAUI ContentView (XAML) to your `Views` folder and call it *SketchWidgetView*.

Modifying the SketchWidgetView.xaml

The contents of the `SketchWidgetView.xaml` file should be modified to the following. Remember that you want to keep your visual tree as simple as possible. You only need to declare the `GraphicsView` itself and no other container controls.

```
<?xml version="1.0" encoding="utf-8" ?>
<GraphicsView
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WidgetBoard.Views.SketchWidgetView"
    StartInteraction="OnGraphicsViewStartInteraction"
    DragInteraction="OnGraphicsViewDragInteraction"
    EndInteraction="OnGraphicsViewEndInteraction" />
```

The `GraphicsView` provides several events that you can subscribe to in order to handle the user's interaction with the surface. You are only interested in the following:

- `StartInteraction`: This is when the user first interacts, so basically when the first touch/mouse click happens.
- `DragInteraction`: This follows from the start and involves the touch/mouse moving around on the surface.
- `EndInteraction`: This is when the user lifts their finger from the screen or mouse button.

When you add these events in the XAML file, it will automatically create some C# code in the `SketchWidgetView.xaml.cs` file that you will expand on shortly.

Modifying the SketchWidgetView.xaml.cs

Visual Studio will have created this file for you already, so you need to open it and modify it to the following.

Note that the types in the event handlers have been shortened (e.g., from `System.Object` to `object`). This is mainly to make it clearer to read.

```
using Microsoft.Maui.Controls;
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;

public partial class SketchWidgetView : GraphicsView,
IWidgetView, IDrawable
{
    public SketchWidgetView()
    {
        InitializeComponent();
        this.Drawable = this;
    }

    public IWidgetViewModel WidgetViewModel
    {
        get => (IWidgetViewModel)BindingContext;
        set => BindingContext = value;
    }

    private void OnGraphicsViewStartInteraction(object sender,
    TouchEventArgs e)
    {
    }

    private void OnGraphicsViewDragInteraction(object sender,
    TouchEventArgs e)
    {
    }

    private void OnGraphicsViewEndInteraction(object sender,
    TouchEventArgs e)
    {
    }
```

```
    public void Draw(ICanvas canvas, RectF dirtyRect)
    {
        throw new NotImplementedException();
    }
}
```

Each of the event handles and the Draw method have the blank or default implementation. Let's build this file up slowly and discuss the key parts as you do so.

First, you need to add the backing fields to store the interactions from the user.

```
private DrawingPath? currentPath;
private readonly IList<DrawingPath> paths = new
List<DrawingPath>();
```

The first event handler to modify is for the StartInteraction event.

```
private void OnGraphicsViewStartInteraction(object sender,
TouchEventArgs e)
{
    currentPath = new DrawingPath(Colors.Black, 2);
    currentPath.Add(e.Touches.First());
    paths.Add(currentPath);
    Invalidate();
}
```

In this method, you first create a new instance of the DrawingPath class, assigning a color and thickness. They can, of course, be expanded to allow selections from the user so they can have custom colors. Next, you add the first touch into the current path so you have your first point of interaction. Then you add the current path to the list of all paths so that they can eventually be rendered on screen. Finally, you call Invalidate, which will trigger the Draw method to be called, and the paths can be drawn.

The next event handler to modify is for the DragInteraction event.

```
private void OnGraphicsViewDragInteraction(object sender,
TouchEventArgs e)
{
    if (currentPath is null)
    {
        return;
    }
    currentPath.Add(e.Touches.First());
    Invalidate();
}
```

In this method, you add the current touch to the current path and again call Invalidate to cause the Draw method to be called.

The final event handler to modify is for the EndInteraction event.

```
private void OnGraphicsViewEndInteraction(object sender,
TouchEventArgs e)
{
    if (currentPath is null)
    {
        return;
    }
    currentPath.Add(e.Touches.First());
    Invalidate();
}
```

This has the exact same implementation as the DragInteraction event handler.

The final set of changes to make is inside the Draw method so you can actually see something on the screen.

```
public void Draw(ICanvas canvas, RectF dirtyRect)
{
    foreach (var path in paths)
    {
        canvas.StrokeColor = path.Color;
        canvas.StrokeSize = path.Thickness;
        canvas.StrokeLineCap = LineCap.Round;
        canvas.DrawPath(path.Path);
    }
}
```

This method loops through all of the paths that you have created from the user interactions, setting the stroke color and size and then drawing the path that was built up by the three event handlers that you just implemented.

Registering Your Widget

The last part in your implementation of the sketch widget is to register your view and view model with the `MauiAppBuilder`. Let's open up the `MauiProgram.cs` file and add the following lines into the `CreateMauiApp` method:

```
WidgetFactory.RegisterWidget<SketchWidgetView, SketchWidgetView
Model>(SketchWidgetViewModel.DisplayName);
builder.Services.AddTransient<SketchWidgetView>();
builder.Services.AddTransient<SketchWidgetViewModel>();
```

Taking Your Widget for a Test Draw

You should be able to run your application on all platforms, add a widget of type *Sketch* to a board, and then interact with the widget to leave a fancy doodle. Figure 15-3 shows the new sketch widget rendered on a board.

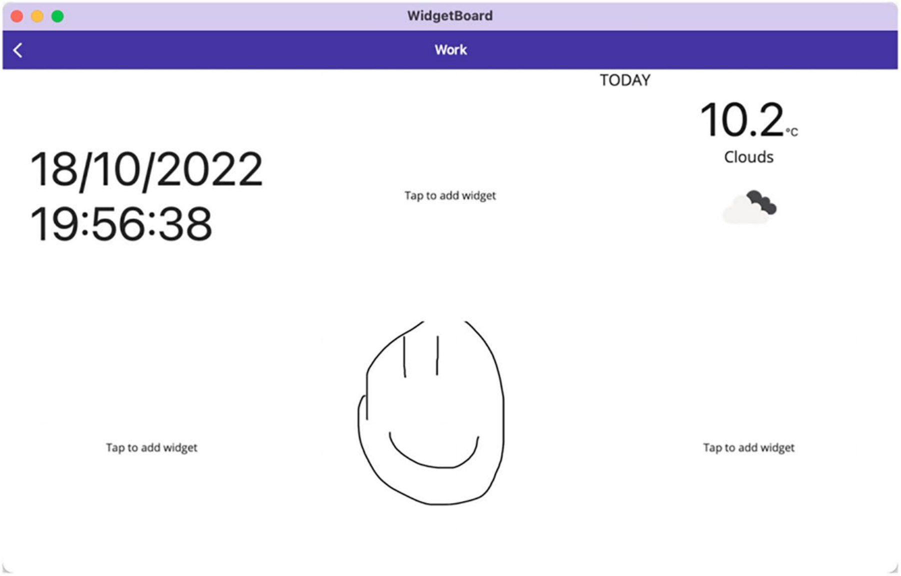

Figure 15-3. *The sketch widget showing my terrible doodling skills running on macOS*

Building an Analog Clock Widget

We have covered how to handle user input and turning that into something rendered on screen. I would like for us to introduce something slightly more complex – an analog clock. Yes, we added the clock widget back in Chapter 4, which displays the digital time, but I would like us to expand on this concept and see how we could convert that information into an analog clock. This will provide some nice exposure to translating and rotating the canvas to hopefully make the maths much more simple.

Creating the AnalogClockWidgetView

As with each of the widget views, you will be creating a XAML-based view. You will be taking a slightly different approach to the *SketchWidgetView* as you will create a view model and rely on that to handle most of the logic.

Add a new **.NET MAUI ContentView (XAML)** to your Views folder and call it *AnalogClockWidgetView*.

Modifying the AnalogClockWidgetView.xaml

The contents of the AnalogClockWidgetView.xaml file should be modified to the following. For the sake of repeating myself, you want to keep your visual tree as simple as possible. You only need to declare the GraphicsView itself and no other container controls.

```
<?xml version="1.0" encoding="utf-8"?>
<GraphicsView
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:WidgetBoard.ViewModels"
    x:DataType="viewmodels:AnalogClockWidgetViewModel"
    x:Class="WidgetBoard.Views.AnalogClockWidgetView" />
```

Modifying the AnalogClockWidgetView.xaml.cs

You can open the *AnalogClockWidgetView.xaml.cs* file and modify the contents to the following:

```
using System.ComponentModel;
using WidgetBoard.ViewModels;

namespace WidgetBoard.Views;
```

```
public partial class AnalogClockWidgetView : IWidgetView
{
    public AnalogClockWidgetView(AnalogClockWidgetViewModel
    clockWidgetViewModel)
    {
        InitializeComponent();
        WidgetViewModel = clockWidgetViewModel;
    }

    private void ClockWidgetViewModelOnPropertyChanged(object?
    sender, PropertyChangedEventArgs e)
    {
        if (e.PropertyName == nameof(AnalogClockWidgetView
        Model.Time))
        {
            Invalidate();
        }
    }

    public IWidgetViewModel WidgetViewModel
    {
        get => (IWidgetViewModel)BindingContext;
        set
        {
            BindingContext = value;

            if (BindingContext is Drawable drawable)
            {
                Drawable = drawable;
            }

            if (BindingContext is INotifyPropertyChaned
            propertyChanged)
            {
```

```
                propertyChanged.PropertyChanged +=
                ClockWidgetViewModelOnPropertyChanged;
            }
        }
    }
}
```

Most of the above should look familiar. I would like to highlight the following additions that may not feel familiar.

The first item is the following line which assigns the view model as the `Drawable` property on `GraphicsView`; this means that the view model will have to implement the `IDrawable` interface because it will be responsible for drawing on the canvas.

```
Drawable = drawable;
```

The next item is the code that subscribes to the `PropertyChanged` event and then calls `Invalidate` on the `GraphicsView` to force the canvas to be redrawn. This allows the `Time` changes in the view model to trigger the canvas to be redrawn.

```
propertyChanged.PropertyChanged +=
ClockWidgetViewModelOnPropertyChanged;
private void ClockWidgetViewModelOnPropertyChanged(object?
sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == nameof(AnalogClockWidgetViewModel.Time))
    {
        Invalidate();
    }
}
```

That concludes the changes required to the view; let's proceed to creating the backing view model.

Creating the AnalogClockWidgetViewModel

The starting point will be to create a new class file in the *ViewModels* folder. Do that and then modify the contents to match the following:

```
namespace WidgetBoard.ViewModels;

public class AnalogClockWidgetViewModel : BaseViewModel,
IWidgetViewModel, IDrawable
{
    public const string DisplayName = "Analog Clock";

    private readonly IDispatcher dispatcher;
    private DateTime time;

    public DateTime Time
    {
        get => time;
        set => SetProperty(ref time, value);
    }

    public int Position { get; set; }

    public string Type => "Analog Clock";

    public AnalogClockWidgetViewModel(IDispatcher dispatcher)
    {
        this.dispatcher = dispatcher;

        SetTime(DateTime.Now);
    }

    public void SetTime(DateTime dateTime)
    {
        Time = dateTime;
```

```
        this.dispatcher.DispatchDelayed(
            TimeSpan.FromSeconds(1),
            () => SetTime(DateTime.Now));
    }

    public void Draw(ICanvas canvas, RectF dirtyRect)
    {
    }
}
```

The above will look very similar to the *ClockWidgetViewModel*; in fact, we could have modified the clock widget to support different rendering modes (e.g., digital or analog); however, for the purpose of the book and keeping the code examples succinct, I have opted to treat them as separate widgets. I have left the `Draw` method empty deliberately for now; I want to explain the maths involved before applying it to the code.

Firstly I would like for our circular clock to render markers for each hour and then also render the hour, minute, and second hands. Given this, we need to consider that

– The second hand will move by a small increment of 6°

– The minute hand will increment by 6° + the amount of seconds that have elapsed

– The hour hand will increment by 30° + the amount of minutes that have elapsed

Figure 15-4 shows the angles required for hour, minute, and second increments.

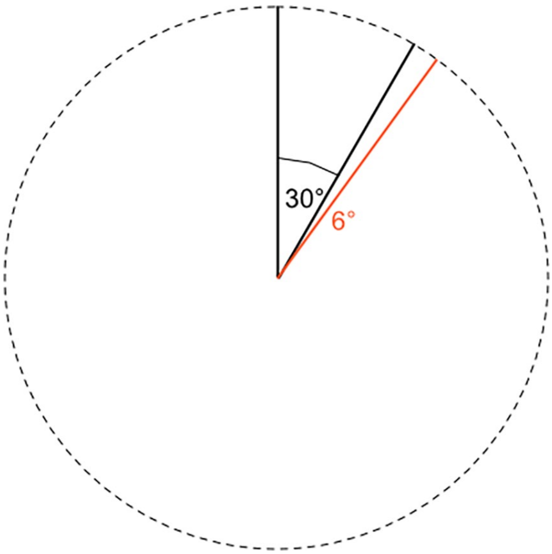

***Figure 15-4.** The angles required for hour, minute, and second increments*

Given the above statements, you will now add in each part incrementally to the `Draw` method in your *AnalogClockWidgetViewModel.cs* file. The first step is to add in some constants and the color-related parts.

```
const int smallIncrement = 6;
const int largeIncrement = 30;

canvas.StrokeSize = 5;
canvas.StrokeColor = App.Current?.PlatformAppTheme == AppTheme.
Dark ? Colors.White : Colors.Black;
```

The above records the small and large angle increments that were detailed in the list above, then the line width has been set, and finally, the line color will either be white or black depending on whether the app is running on a system with dark or light mode.

The next step is to render the hour markers on the clock. Interactions with the canvas work in a similar way to general user interface building; the earlier items are added or drawn the lower down in the visual tree they are; therefore, if you rendered two things in the same space, it would be the last thing rendered that would be visible. Let's add the code to render the markers.

```
var radius = dirtyRect.Size.Height / 2;

canvas.Translate(dirtyRect.Center.X, dirtyRect.Center.Y);
var hourMarkerLength = dirtyRect.Size.Height / 10;

for (var i = 1; i <= 12; i++)
{
    canvas.Rotate(largeIncrement);
    canvas.DrawLine(0, -(radius - hourMarkerLength), 0,
    -radius);
}
```

The above code first determines the radius of the clock being half the height of the widget; you could introduce some padding here if you wanted or possibly check if the width is smaller than the height and take that value. The next step is to perform a translation; this essentially means you are moving where the origin is on the canvas – your code has now moved the origin from the top left to the center of the canvas. Then inside the `for` loop, the code will rotate the canvas by a `largeIncrement`, which is 30°, and then draw a line to represent the marker. That is all the code required to render the markers, so let's proceed to rendering the hour hand using the above statement: *"the hour hand will increment by 30° + the amount of minutes that have elapsed"*.

```
const float minuteIncrement = smallIncrement / 60f;
var hourAngle = (Time.Hour * largeIncrement) + Time.Minute *
minuteIncrement;
canvas.Rotate(hourAngle);
```

```
canvas.DrawLine(0, -5, 0, -(dirtyRect.Size.Height / 5));
canvas.Rotate(-hourAngle);
```

The first part is to calculate the increment to apply to the "amount of minutes that have elapsed" part; this is the smallIncrement divided by the number of minutes in an hour. Then the code calculates the angle to rotate the canvas by, draw the line, and then rotate back to 0. You could calculate the difference between the hour hand and the minute hand, but sometimes I find a well-intentioned piece of code like the above makes it easier for others to come in and read.

One key part I should highlight is the use of negative numbers when drawing the lines – this is because we are treating 12:00 as 0 degrees and to render that without any orientation from our origin to the center of the canvas, we need to draw a line upward.

The next code for the minute hand should look very similiar to the hour hand.

```
const float secondIncrement = smallIncrement / 60f;
var minuteAngle = (Time.Minute * smallIncrement) + Time.Second
* secondIncrement;
canvas.Rotate(minuteAngle);
canvas.DrawLine(0, -5, 0, -(dirtyRect.Size.Height / 3));
canvas.Rotate(-minuteAngle);
```

You have just replaced the addition of minutes elapsed with seconds elapsed and made the line slightly bigger.

Finally, the code for the second hand can be added as follows:

```
canvas.StrokeSize = 3;

var secondAngle = Time.Second * smallIncrement;
canvas.Rotate(secondAngle);
canvas.DrawLine(0, -5, 0, -(dirtyRect.Size.Height / 3));
canvas.Rotate(-secondAngle);
```

There are no extra increments to add here, so it is a simpler calculation for the angle. This concludes the code required to render the clock; let's proceed to registering the widget so it can be added to a user's board.

Registering Your Widget

The last part in your implementation of the sketch widget is to register your view and view model with the `MauiAppBuilder`. Let's open up the `MauiProgram.cs` file and add the following lines into the `CreateMauiApp` method:

```
WidgetFactory.RegisterWidget<AnalogClockWidgetView, AnalogClock
WidgetViewModel>(AnalogClockWidgetViewModel.DisplayName);
builder.Services.AddTransient<AnalogClockWidgetView>();
builder.Services.AddTransient<AnalogClockWidgetViewModel>();
```

Taking Your Widget for a Test Draw

You should be able to run your application on all platforms, add a widget of type *Analog Clock* to a board, and then watch as the seconds tick by. Figure 15-5 shows the new analog clock widget rendered on a board.

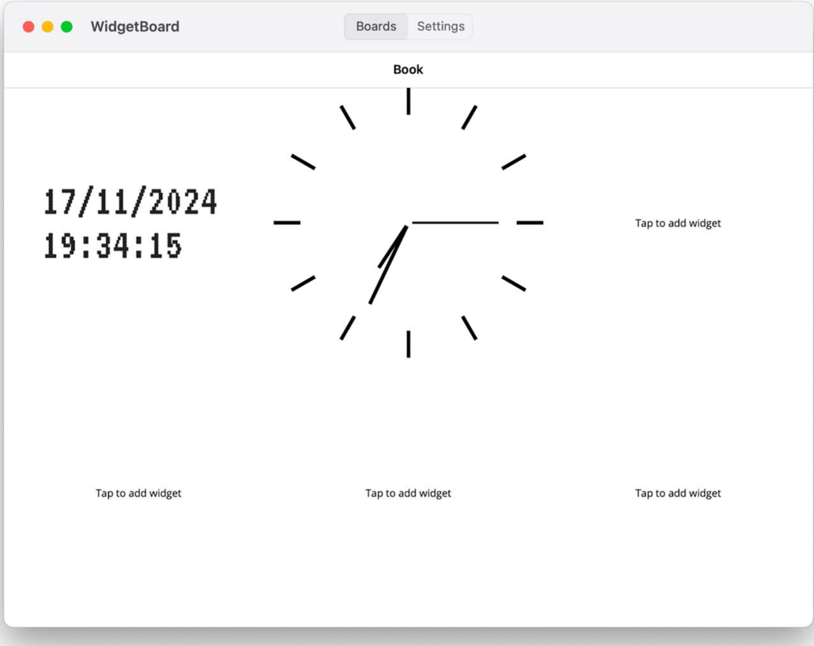

Figure 15-5. *The application showing both the original clock and new analog clock widgets*

Summary

In this chapter, you have

- Learned what .NET MAUI Graphics is
- Gained an insight into some of the power provided by .NET MAUI Graphics
- Built your own sketch widget with the .NET MAUI `GraphicsView` control
- Taken the Graphics APIs further to also create an analog clock widget

In the next chapter, you will

- Explore the concepts of distributing your application
- Learn about concepts like continuous integration and continuous delivery to improve your development processes
- Learn about linking, what it is, and how it can benefit/hinder you
- Learn why it is important to collect analytical and crash information
- Explore why you might want to consider obfuscating your code

Source Code

The resulting source code for this chapter can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch15`.

Extra Assignment

Perhaps you can think of another concept where you can use .NET MAUI Graphics – maybe the chart control idea I discussed or even just showing the battery level in a widget or other device information.

Source Code

I would love for you to have an attempt at this extra assignment, but I have also provided the source code. The source code for this extra assignment can be found on the GitHub repository at `https://github.com/Apress/Introducing-.NET-MAUI-2nd-ed/tree/main/ch15-extra`.

CHAPTER 16

Releasing Our Application

Abstract

Once you have built your application, you need to get it to your users. There are many ways to achieve this. You can publish a release build and ship it directly to your customers or you can make use of the stores that each platform provider offers.

Shipping directly to an end customer can sometimes be the best option, such as when you are building an internal application and you don't want it to be publicly accessible.

Most often the recommended way to ship applications to users is to go through the stores provided by each platform provider (e.g., App Store from Apple, Play Store from Google, and Microsoft Store from Microsoft). This does involve agreeing to terms and conditions, and these providers take a percentage of any income you make. There are many benefits that justify paying the fees. They provide trusted platforms for users to find and download your applications. The store will provide a much wider reach for your intended audience. The store also manages the ability to provide updates seamlessly.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_16

This chapter has been split into two key sections: how you would go about distributing your application and additional improvements you can make to boost performance and reduce application size.

Distributing Your Application

The aim of this section is not to give a step-by-step guide on distributing to each of the stores mentioned above. Initially, I wanted to provide this information, but the details around doing so have changed numerous times during the time it has taken to write this book. For this reason alone, I will defer to the platform providers and Microsoft's documentation on how to achieve distribution. What this chapter will cover is details around distributing applications, how to build a relevant binary for each store, why you need to do it, and some of the common issues that crop up during the process.

One very important thing to note is that apps built with .NET MAUI follow the same rules and common issues that native applications follow. Therefore, when encountering issues with each specific store, sometimes a search engine will yield better results if you omit the .NET MAUI part.

I would like to highlight now that for each time you will be calling `dotnet publish` within this chapter, you will need to follow these steps based on the operating system that you are using.

macOS

1. Open the **Terminal** application.
2. Enter the following command and then press return:

```
dotnet publish /path/to/WidgetBoard.csproj
```

Windows

1. Open the **Command Prompt** application.
2. Enter the following command and then press return:

```
dotnet publish /path/to/WidgetBoard.csproj
```

Android

Android has the biggest mobile user base. However, given the model it follows of allowing manufacturers to customize the Android operating system as well as providing varying sets of hardware, it can be the most problematic.

An *Android Package*, or APK for short, is the resulting application file that runs on an Android device. If you wish to provide a mechanism to download this file (e.g., a website or file share), users can side-load the application onto their Android device. This is not recommended in the public domain because it can be very difficult to trust the packages that are freely downloaded from the Internet.

When you wish to distribute using the Google Play Store, you are required to build an *Android App Bundle*, or AAB for short. It contains all of the relevant files needed to compile an APK ready for installation on a user's device.

Essentially you build an Android App Bundle, sign it with a specific signing key that you own, and upload the bundle to Google Play. Google uses this bundle when a user comes to download your application and compiles a specific APK for that device. This is the way to do things now. If you have worked with Android apps in the past, you may recall building the APK yourself. This runs into the issue that the APK is architecture specific, and in the current market where there are multiple architectures supported by the various Android devices, you can end up with an application size that is the sum of the number of architectures multiplied by the actual size (e.g., if there are four architectures and the application size is 25 MB, the resulting APK is 100 MB).

Generating Your Android Application

This section will focus on how to generate the application using the command-line tooling provided by .NET. The main reason for this is that I believe understanding how to achieve it this way will make it relatively straightforward to apply it to other scenarios such as including in your CI/CD pipelines.

```
dotnet publish /path/to/WidgetBoard.csproj --framework
net9.0-android
```

It is worth highlighting that you can provide values for any properties that can be added into a csproj file. For example, if you look in your WidgetBoard.csproj file, you will see an entry for `<ApplicationVersion>1</ApplicationVersion>`. This is an excellent example because it is highly unlikely that you will always build an application with the version of 1, and it is also unlikely that you will control the version number inside the csproj file. In order to provide the value to a build process, you can provide it as follows:

```
-p:ApplicationVersion=”12”
```

Therefore, the above publish command could look as follows:

```
dotnet publish /path/to/WidgetBoard.csproj --framework net9.0-
android -p:ApplicationVersion=”12”
```

Again this might not feel very helpful using a fixed number, but if you imagine generating a number during your CI/CD pipeline build, then you could provide that in as the value and then you have dynamic version generation based on the version of the source being compiled. Furthermore, it leads us nicely onto other properties that we can and must supply when generating an Android application – signing properties.

Signing Your Android Application

In order to distribute the application via the Google Play Store, you will need to sign the application built using the `dotnet publish` command that we have already covered. The reason for this is to allow Google to verify that it is you uploading the builds and not someone else. With this in mind, you will need to keep your signing details in a safe place!

In order to sign your Android application, you'll need to use a signing key from your keystore. A *keystore* is a database of security certificates that's created by using keytool from the Java Development Kit (JDK).

To create a keystore file, the following steps can be followed:

1. Open the command prompt or terminal application on your computer.
2. Navigate to the folder of your project.
3. Run the *keytool* tool with the following parameters:

```
keytool -genkeypair -v -keystore {filename}.keystore -alias
{keyname} -keyalg RSA -keysize 2048 -validity 10000
```

The values above inside the `{ }` characters should be replaced with values that you decide upon. Once you have created your keystore and stored it somewhere safe, you can provide it to the `dotnet publish` command-line tool and generate an AAB file that can be uploaded to the Google Play Store. The following shows the full command:

```
dotnet publish --framework net9.0-android
    -p:AndroidKeyStore=true
    -p:AndroidSigningKeyStore="{filename}.keystore"
```

```
-p:AndroidSigningKeyAlias={keyname}
-p:AndroidSigningKeyPass="{keypassword}"
-p:AndroidSigningStorePass="{keypassword}"
```

Once you have run the above command, you will find that the tooling has created a file named *com.tinysoft.widgetboard-Signed.aab* under the *WidgetBoard/bin/Release/net9.0-android/publish* folder.

If you plan to use these details in your CI/CD pipelines, then I would strongly recommend you look into secure options such as GitHub Secrets, or whatever is appropriate on the platform that you are using.

Additional Resources

Both Microsoft and Google provide documentation on how to distribute applications via the Google Play Store. See the following links.

- Microsoft: How to publish an application ready for the Play Store, `https://learn.microsoft.com/dotnet/maui/android/deployment/overview`
- Google: How to upload your application to the Play Store, `https://developer.android.com/studio/publish/upload-bundle`
- It is also worth noting that other stores/platforms provide the ability to distribute, install, and run Android applications. Amazon devices such as the Kindle Fire are built on top of Android and allow the running of Android applications. Amazon provides its own store, details of which can be found at `https://developer.amazon.com`.

iOS

iOS and macOS are considered really painful when dealing with distributing and signing. Having spent several years going through this pain, I want to break down the key concepts to hopefully reduce the pain that you might experience. Thankfully the Apple tooling has come a long way since I started building mobile apps in 2007, so you don't have to relive all those painful memories.

The following sections cover the development and application settings you will need to create on the Apple developer website at `https://developer.apple.com`.

Certificate

You need to generate a certificate on the machine that will build your application. Most documentation takes you through the complex scenarios of creating a Certificate Signing Request and then uploading to Apple. There is actually a far simpler way by using Xcode. The following steps can help to achieve this:

- Click the **Xcode** main menu.
- Click **Settings**.
- Click the **Accounts** tab.
- Click the **Manage Certificates** button, shown in Figure 16-1.

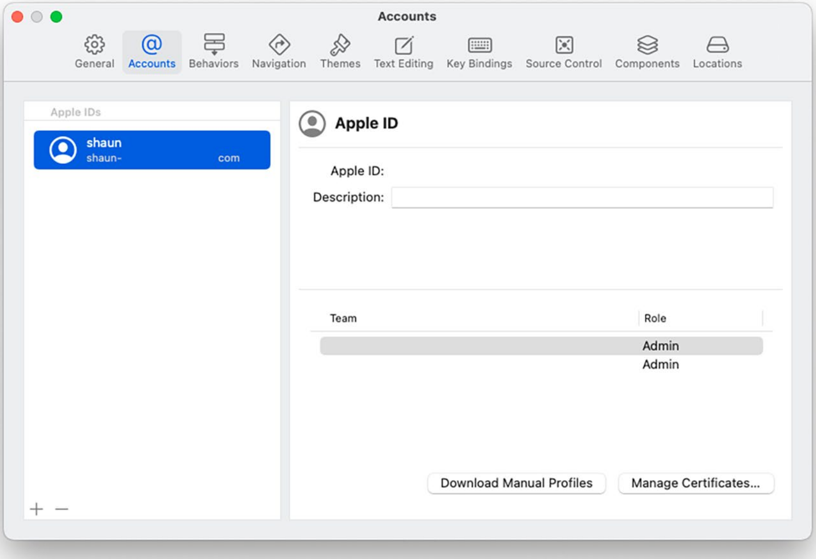

Figure 16-1. *Apple settings screen showing how to manage certificates*

Identifier

This represents your application. It requires you to define unique details to identify the application that will be exposed to the public store as well as defining what capabilities your application requires. Note that the value you provide to Apple needs to match the value set in your project file under `<ApplicationId></ApplicationId>` in your csproj file (WidgetBoard.csproj in our case).

Capabilities

iOS applications run under a sandboxed environment. Apple provides a set of App Services that can be utilized by your applications and enhance its capabilities. Capabilities include services like in-app purchasing, push notifications, Apple Pay, and such. The use of these services needs to be defined at compile time, and the usage of them will be reviewed when you upload your application to Apple for review. Therefore, it's important that you make sure you only have the ones you need. Don't worry, though; a failure here will give a fairly useful error message and can be a relatively easy fix. More information can be found at `https://developer.apple.com/documentation/xcode/capabilities`.

Changes to the capabilities of your application will invalidate your provisioning profiles so they will need to be edited in the `https://developer.apple.com` portal to update them with the newer capabilities.

Entitlements

Entitlements tie in closely with capabilities and allow you to configure settings during compilation. You need to add an `Entitlements.plist` file to your application and then add the relevant entries for the configuration. Information on how to configure this can be found at `https://learn.microsoft.com/dotnet/maui/ios/deployment/entitlements`.

Provisioning Profiles

Provisioning profiles determine how your application will be provisioned for deployment. There are two main types:

- Development: This is what you need when running a debug build of your application on your own device.
- Distribution: This is required for the release builds when you ship to the App Store.

A common issue around provisioning profiles is when trying to run the application on your device and the tooling reports back *Unable to deploy app to this device, no provisioning profiles were found*. When observing this, a good starting point is to double-check that you have the provisioning profile installed and whether the profile has been invalidated by changing any capabilities.

Generating Your iOS Application

This first part should look very similar to the Android arguments except we provide a different Target Framework Moniker or TFM for short. The TFM we provide to publish an iOS application is `net9.0-ios`, and you will have spotted that this value exists inside your *WidgetBoard.csproj* file in the `<TargetFrameworks>` element.

```
dotnet publish /path/to/WidgetBoard.csproj --framework net9.0-ios
```

In order to sign the iOS application, you need to provide two additional arguments:

- `-p:CodesignKey=""`: This is the name of the certificate that you will have created using Figure 16-1.
- `-p:CodesignProvision=""`: This is the name of the provisioning profile that you create on the Apple developer website. Note that this will want to be of type distribution.

Using the above details, we can now publish a signed iOS application using the following example:

```
dotnet publish /path/to/WidgetBoard.csproj
    --framework net9.0-ios
    -p:CodesignKey="Apple Distribution: Company Name
    (ABCDEF12345)"
    -p:CodesignProvision="WidgetBoard Distribution"
```

Note that if you run this command now, you should see two warnings and zero errors reported. If you were proactive during the building of the application and solved the warnings, then great work! I haven't opted to ignore; in fact, I wanted to show how they can manifest into some less clear warnings when we turn on full trimming later on in this chapter. For clarity, the two warnings that I am referring to are

1>BoardDetailsPage.xaml(54,38): Warning XC0045 XamlC: Binding: Property "IsChecked" not found on "WidgetBoard.ViewModels. BoardDetailsPageViewModel".

1>FixedBoardPage.xaml(59,21): Warning XC0045 XamlC: Binding: Property "Text" not found on "WidgetBoard.ViewModels. FixedBoardPageViewModel".

Don't worry, we will fix these in the "Trimming" section after we look at the impact they have on trimming itself.

Once you have run the above command, you will find that the tooling has created a file named *WidgetBoard.ipa* under the *WidgetBoard/bin/ Release/net9.0-ios/ios-arm64/publish* folder.

Additional Resources

Both Microsoft and Apple provide documentation on how to distribute applications via the Apple App Store.

- Microsoft: How to publish an application ready for the App Store, `https://learn.microsoft.com/dotnet/maui/ios/deployment/overview`
- Apple: How to upload your application to the App Store, `https://developer.apple.com/app-store/`

macOS

When distributing your .NET MAUI application for macOS, you can generate a `.app` or a `.pkg` file. A `.app` file is a self-contained app that can be run without installation, whereas a `.pkg` is an app packaged in an installer.

Generating Your macOS Application

This part should start to feel familiar; initially we are just changing the TFM to `net9.0-maccatalyst`.

```
dotnet publish /path/to/WidgetBoard.csproj --framework net9.0-
maccatalyst
```

In order to build the macOS application that can be distributed via the Apple App Store, you will need to generate a .pkg file. In order to do this, you will need to provide the following additional build properties:

- `-p:CreatePackage=true`: This will tell the tooling to create a .pkg file.
- `-p:EnableCodeSigning=true`: This will enable code signing for the application.
- `-p:PackageCodeSigning=true`: This will enable code signing for the .pkg file that is created.
- `-p:CodesignKey=""`: This is the name of the certificate that you will have created using Figure 16-1.
- `-p:PackageSigningKey=""`: This is also the name of the certificate that you will have created using Figure 16-1.
- `-p:CodesignProvision=""`: This is the name of the provisioning profile that you create on the Apple developer website. Note that this will want to be of type distribution.

Using the above details, we can now publish a signed iOS application using the following example:

```
dotnet publish /path/to/WidgetBoard.csproj --framework net9.0-
maccatalyst -p:CreatePackage=true -p:EnableCodeSigning=true
-p:EnablePackageSigning=true -p:CodesignKey="Apple
Distribution: Company Name (ABCDEF12345)" -p:CodesignProvision
="WidgetBoard Distribution"-p:PackageSigningKey="3rd Party Mac
Developer Installer: Company Name (ABCDEF12345)"
```

Once you have run the above command, you will find that the tooling has created a file named *WidgetBoard-1.0.pkg* under the *WidgetBoard/bin/Release/net9.0-maccatalyst/publish* folder.

Additional Resources

Both Microsoft and Apple provide documentation on how to distribute applications via the Apple App Store.

- Microsoft: How to publish an application ready for the App Store, https://learn.microsoft.com/dotnet/maui/macos/deployment/overview
- Apple: How to upload your application to the App Store, https://developer.apple.com/macos/distribution/

Windows

When distributing your .NET MAUI app for Windows, you can publish the app and its dependencies to a folder for deployment to another system. Publishing a .NET MAUI app for Windows creates an unpackaged application by default, making it possible for you to create an installer for your application. If you wish to distribute your application via the Microsoft Store, then you need to modify your application to be published as a packaged application (MSIX format).

MSIX is a Windows app package format that provides a modern packaging experience to all Windows apps.

In order to build a packaged application, you can open your WidgetBoard.csproj file and modify the following line (change in **bold**):

```
<WindowsPackageType>MSIX</WindowsPackageType>
```

Generating Your Windows Application

In order to generate an MSIX package, you will first need to create a self-signed certificate which can be done as follows:

1. Open a PowerShell terminal and navigate to the directory with your project.

2. Use the `New-SelfSignedCertificate` command to generate a self-signed certificate:

   ```
   New-SelfSignedCertificate -Type Custom
   -Subject "CN=<PublisherName>" -KeyUsage
   DigitalSignature -FriendlyName "My temp dev
   cert" -CertStoreLocation "Cert:\CurrentUser\My"
   ```

3. Use the following PowerShell command to query the certificate store for the certificate that was created:

   ```
   Get-ChildItem "Cert:\CurrentUser\My" | Format-
   Table Thumbprint, Subject, FriendlyName
   ```

4. Copy the Thumbprint to your clipboard as you will use it in the `dotnet publish` command.

You can run the command as follows:

```
dotnet publish /path/to/WidgetBoard.csproj --framework net9.0-
windows -p:PackageCertificateThumbprint=ABC123
```

Where `ABC123` will be the value of the Thumbprint for the certificate that you just created.

Once you have run the above command, you will find that the tooling has created a file named *WidgetBoard.msix* under the *WidgetBoard/bin/Release/net9.0-windows/publish/win10-x64* folder.

Additional Resources

Microsoft provides documentation on how to distribute applications via the Microsoft Store.

- Microsoft: How to publish an application ready for the App Store, https://learn.microsoft.com/dotnet/maui/windows/deployment/overview
- Microsoft: How to upload your application to the Microsoft Store, https://developer.microsoft.com/microsoft-store/

This concludes the section on how to generate binaries that can be uploaded to each platform store. Let's proceed to taking a look at how applications can be optimized.

Optimizing Your Application

Many issues can crop up when you make the jump from a debug build running on a simulator, emulator, or physical device to building a release build ready to run on an end user's machine.

Following Good Practices

Each of the platform-specific sections prior to this one contained information or links to resources that show how to deploy your applications to each platform provider's public store. This is all great, but one key detail that is lacking is the use of continuous integration and continuous delivery (CI/CD) in order to provide a clean environment that can reliably produce a build that can be deployed.

Continuous integration (CI) is the practice of merging all developers' working copies to a shared mainline.

Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, without doing so manually. It aims at building, testing, and releasing software with greater speed and frequency. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery.

Both concepts are usually considered together as they help to make it a far smoother experience when working in a team. I was there in the early stages of learning and building apps and I neglected this part. If I could go back and tell a much younger Shaun some advice, it would be to get this part set up and early in the development process. Thanks to the .NET CLI that is available to us, the setup to provide the necessary steps is straightforward. On top of that, tools like GitHub, Azure DevOps, TeamCity, and others will likely provide some level of out-of-the-box support for this.

We covered earlier in this chapter how each of the applications can be built with the .NET CLI; for example, Android can be built:

```
dotnet publish --framework net9.0-android
```

I am using `net9.0` here because my application is built against .NET 9.0. If you are working against a different version of .NET, replace `net9.0` with your chosen version. If you are unsure what version you are using, open your csproj file and look at the value inside the `<TargetFrameworks></TargetFrameworks>` tags.

There are more required arguments to pass to the build, which involve signing key passwords and more, but this shows how easily this can be added to a set of automated steps that run each time code is committed or a merge request is opened.

You should also consider the testing that you added in Chapter 13 and see how this can also be incorporated into a CI environment.

```
dotnet test
```

This is far simpler than the publishing step. Running the tests in a CI environment really should be considered a critical set of criteria when building any application. The safety net that this provides in making sure your changes do not unintentionally break other bits of functionality alone makes it worthwhile.

Performance

Android has always been one of the slower platforms when building mobile applications. Don't get me wrong; the applications can perform well on the higher-end devices, but Android devices come in a wide range of specifications, and typically in the business environment, it is the cheaper devices that get bought in bulk and are expected to perform well. There are some concepts that you should consider when publishing your Android applications in order to boost the performance of your applications.

Startup Tracing

There are some extra steps that you can do in order to boost the startup times of your Android applications. Startup tracing essentially profiles an application when it starts to determine what libraries and other initializations are required so when you release the application, it will benefit from a faster startup time. It is worth noting that boosting the startup time can result in an increase in application size, so I recommend playing around with the settings to find the right balance for your application.

Microsoft has published two great blog posts on how startup tracing can be configured, the improvements it makes, and how the application can be affected:

- https://devblogs.microsoft.com/dotnet/dotnet-7-performance-improvements-in-dotnet-maui/
- https://devblogs.microsoft.com/xamarin/faster-startup-times-with-startup-tracing-on-android/

Image Sizes

One thing that can perform really poorly is the use of images that do not match the dimensions in which they need to be rendered on screen. For example, an image that displays at 100×100 pixels in the application really should be that size when supplied. If you were to render an image that was actually 300×300 pixels, it will not only look poor on the device due to scaling, but it will slow the application down. Plus, it involves storing an image that is bigger than really needed. Therefore, make sure that your images are correctly sized to gain the best experience when rendering them.

Use of ObservableCollection

A lot of common coding examples show how to bind an `ObservableCollection` to the `ItemsSource` property of a control. This can have its uses, but it can have a big performance overhead. The reason is that each time an element is added to the collection, a UI update will be triggered because the control is monitoring for changes against the `ObservableCollection`. If you do not need live updating items in a collection, it is typically much faster to use a List and simply raise the `PropertyChanged` event from `INotifyPropertyChanged` instead.

Let's take a look at the code you added in Chapter 9 and see how it can be improved:

```
public ObservableCollection<Board> Boards { get; } = new
ObservableCollection<Board>();
public void LoadBoards()
{
    var boards = this.boardRepository.ListBoards();
    foreach (var board in boards)
    {
        Boards.Add(board);
    }
}
```

You can improve the performance of the above code by implementing it with a `List` as follows:

```
private IList<Board> boards;
public IList<Board> Boards
{
    get => this.boards;
    private set => SetProperty(ref this.boards, value);
}
```

```
public void LoadBoards()
{
    Boards = this.boardRepository.ListBoards();
}
```

This new code will result in the UI only being updated once rather than once per each board that is added to the `Boards` collection.

Additional Resources

We have covered a large number of techniques and good practices to follow throughout the course of this book in order to avoid a poorly performing application. Microsoft does provide guidance on how to detect issues and resolve them here: `https://learn.microsoft.com/dotnet/maui/deployment/performance`.

Trimming

While devices these days do tend to offer generous amounts of storage space, it is still considered a very good practice to minimize the amount of memory your apps really consume, especially when considering mobile devices that have limited data networks in order to download the apps.

What Is Trimming?

Trimming is performed by the tooling to remove unused code from compiled assemblies. This helps to reduce the size of your applications by trimming out any unused parts of libraries that you use. By default, trimming is set to partial trimming, which means that only the .NET MAUI assemblies will be scanned and have any unused code removed; this is because those assemblies have been updated to be trim safe.

Trimming is a highly complex topic, and while you will be covering how to fix some warnings the trimmer reports, I feel like I am only really scratching the surface. For further reference, I recommend checking out the Microsoft documentation at `https://learn.microsoft.com/dotnet/maui/deployment/trimming`.

The trimmer provides the fantastic ability to reference the full .NET Base Class Library (BCL) so when you compile your application ready for distribution, it will only include the parts of that BCL that you actually reference and use within your application.

One term used to describe part of the trimming process is *tree shaking*. I believe the analogy holds up fairly well, if you imagine the tree (your application code) is connected to branches (libraries) which are connected to leaves or fruit (functions/classes inside the libraries). It will only be the leaves and fruit that your application directly depends on and references that won't fall from the tree when it is shaken by the .NET compiler.

As you can imagine, if the Trimmer is unable to detect that something is really used in your application and it is removed, things can go very wrong at runtime. Your application will most likely crash when it tries to use a type that isn't included in your build.

This can quite often happen when only referring to types in XAML. As I covered in Chapter 5, the XAML compiler isn't as powerful as the C# compiler, and it can miss scenarios. Since the release of .NET 9.0, a lot of effort has been made to bridge the gap in terms of functionality or at least the detection of used types. With this point in mind, we should revisit the topic of compiled bindings from Chapter 5; by default, in .NET MAUI 9.0+, the code base's warnings will be reported if you do not use compiled bindings; this is to prevent the scenario that I just mentioned. Of course these are just warnings and won't stop you from building and even shipping your application, but please heed this warning and make sure to clear all of those warnings up. Don't worry, as part of this section, we will work through concrete examples of how to fix these issues.

We have covered XAML and its limitations, but another key feature to avoid is Reflection, or if you do use it, be careful to make sure that the functionality the code is reflecting over will not be trimmed out. Not only can it trick the compiler into not realizing APIs are used but it can also not perform well.

It is worth considering that some third-party packages that you end up using in your applications may not be trimmer safe. For this reason, the default setting of **partial** is set. This means that only the assemblies provided by Microsoft will be linked because they are built to be trimmer safe. In an ideal world, the third-party libraries would also be trimmer safe, but I can safely say that the people building these fantastic packages are already spread thin building them, so if it is something that you really require, I strongly urge you to investigate helping them provide it or sponsoring the people that build it to help them.

Enable Trimming

You can turn on full trimming with

```
<TrimMode>full</TrimMode>
```

If you make the change above in the *WidgetBoard.csproj* file and then run the `dotnet publish` command again, you should see the following warnings being reported. Note that I have run the command for `net9.0-ios,` but you could run it against any target you desire.

Warning IL2087

The first warning that I wanted to highlight is reported as follows:

*/Users/shaunlawrence/Documents/work/projects/introducing-maui-samples/second-edition/WidgetBoard/WidgetBoard/**MauiProgram.cs**(95,9): Trim analysis **warning IL2087**: WidgetBoard.MauiProgram.AddPage<TPage,TViewModel>(IServiceCollection, String): 'serviceType' argument does not satisfy 'DynamicallyAccessedMemberTypes.*

PublicConstructors' in call to 'Microsoft.Extensions.DependencyInjection.ServiceCollectionServiceExtensions.AddTransient(IServiceCollection, Type)'. The generic parameter 'TPage' of 'WidgetBoard.MauiProgram.AddPage<TPage,TViewModel>(IServiceCollection, String)' does not have matching annotations. The source value must declare at least the same requirements as those declared on the target location it is assigned to.

The warning is telling us that the `AddTransient` method provided by .NET MAUI declares the `DynamicallyAccessedMemberTypes.PublicConstructors`, and because our method `AddPage` calls `AddTransient`, we need to preserve this declaration in order to make sure that the method and related types are not trimmed out. Given this, we can look to apply this change.

Open the *MauiProgram.cs* file and make the following change in **bold**.

```
private static void AddPage<[DynamicallyAccessedMembers
(DynamicallyAccessedMemberTypes.PublicConstructors)] TPage,
[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.
PublicConstructors)] TViewModel>(
```

This will resolve the warning and make sure that trimming does not introduce any unexpected behavior.

Warning IL2026

This warning is reported multiple times, and while the warning message might be similar, the fix is slightly different for each example that we will work through. The first warning is reported as

*ILLink : Trim analysis **warning IL2026**: WidgetBoard.ViewModels.BoardDetailsPageViewModel: Using member 'Microsoft.Maui.Controls.QueryPropertyAttribute.QueryPropertyAttribute(String, String)' which has 'RequiresUnreferencedCodeAttribute' can break functionality when trimming application code. Using QueryPropertyAttribute is not trimming friendly and might not work correctly. Implement the IQueryAttributable* *interface instead. https://learn.microsoft.com/dotnet/maui/fundamentals/shell/navigation#process-navigation-data-using-a-single-method*

Yes, I know that we introduced the use of the `QueryProperty` attribute earlier on in this book. I could have left it out, but I wanted to show how to fix a number of trimming warnings and this is a great example. Also at the point of writing, if you use a search engine to look up how to pass data between view models, you will find blog posts referring to the `QueryProperty` attribute; therefore, I believe it is important to explain the detail here and show why we shouldn't use it. Thankfully the message provided by the analyzer tells us how to fix it, but the following section will show how to fix it step by step.

The first step is to open the *BoardDetailsPageViewModel.cs* file.

Then you will want to delete the following line from the file:

```
[QueryProperty(nameof(BoardCreatedCompletionSource), "Created")]
```

Then you will want to make the class implement the `IQueryAttributable` interface; see below with change in **bold**:

```
public class BoardDetailsPageViewModel : BaseViewModel,
IQueryAttributable
```

Finally, you can add the `ApplyQueryAttributes` method that is required by the `IQueryAttributable` interface as follows:

```
public void ApplyQueryAttributes(IDictionary<string,
object> query)
{
    BoardCreatedCompletionSource = query["Created"] as
    TaskCompletionSource<Board?>;
}
```

These changes will resolve the warning.

Warning IL2026

This is the second occurrence of the IL2026 warning that we should be observing and fixing. You will notice that the error message is different to the previous section despite it being the warning; it is reported as

*/Users/shaunlawrence/Documents/work/projects/introducing-maui-samples/second-edition/WidgetBoard/WidgetBoard/obj/Release/net9.0-ios/ios-arm64/Microsoft.Maui.Controls.SourceGen/Microsoft.Maui.Controls.SourceGen.CodeBehindGenerator/**Pages_BoardDetailsPage.xaml.sg.cs**(30,3): Trim analysis **warning IL2026**: WidgetBoard.Pages.BoardDetailsPage.InitializeComponent(): Using member 'Microsoft.Maui.Controls.Xaml.Internals.XamlTypeResolver.XamlTypeResolver(IXmlNamespaceResolver, Assembly)' which has 'RequiresUnreferencedCodeAttribute' can break functionality when trimming application code. Loading XAML at runtime might require types and members that cannot be statically analyzed. Make sure all of the required types and members are preserved.*

You may also recall me saying that there are two warnings reported in the "Generating Your iOS Application" section; this is where we take a closer look at that. For the sake of repeating myself, you can see that the original warning is as follows:

1>BoardDetailsPage.xaml(54,38): Warning XC0045 XamlC: Binding: Property "IsChecked" not found on "WidgetBoard.ViewModels.BoardDetailsPageViewModel".

The main reason I am repeating this here is to show the difference between the warning you see before setting `TrimMode` to `Full` and after. I want to highlight that the warning before actually tells you how to fix the issue whereas I am not sure I would know where to start with the new warning. With this in mind, I would fully recommend working through all warnings in your application prior to enabling features like trimming or NativeAOT. Now that we have some valuable context in the original warning message, let's proceed to fixing it.

The way to resolve this warning is to open the *BoardDetailsPage.xaml* file and make the following change (in **bold**):

```
<VerticalStackLayout IsVisible="{Binding
IsChecked, Source={x:Reference FixedRadioButton},
x:DataType=RadioButton}">
```

This change tells the XAML compiler what type is being provided to the `Source` property and means that a compiled binding can be created. This time the type being provided is of type `RadioButton`.

There is also a second instance of this warning to fix. I am only including the warning which is reported prior to enabling full trimming mode because it tells us how to fix it.

1>FixedBoardPage.xaml(59,21): Warning XC0045 XamlC: Binding: Property "Text" not found on "WidgetBoard.ViewModels. FixedBoardPageViewModel".

The way to resolve this warning is to open the *FixedBoardPage.xaml* file and make the following change (in **bold**):

```
<Picker
    ItemsSource="{Binding AvailableWidgets}"
    SelectedItem="{Binding SelectedWidget}"
    SemanticProperties.Description="{Binding Text,
Source={x:Reference SelectTheWidgetLabel}, x:DataType=Label}"
```

This change tells the XAML compiler what type is being provided to the `Source` property and means that a compiled binding can be created. This time the type being provided is of type `Label`.

This now concludes the changes required to enable full trimming mode. You won't be clear of trimmer warnings, but the changes required to fix them are the same for when enabling NativeAOT support which we will now cover.

Ahead-of-Time Compilation

Ahead of Time or AOT for short is the process. At the time of writing, AOT is not supported for Android, but all other platforms are supported.

We covered back in Chapter 1 how .NET MAUI applications run on the Mono runtime on Android, iOS, and macOS. By enabling AOT in a .NET MAUI application, our applications will run on an entirely different runtime – the NativeAOT runtime.

Enable NativeAOT

In order to enable AOT compilation in your applications, you can add the following to your project. In fact, let's add it to the *WidgetBoard.csproj* file.

```
<PropertyGroup>
    <IsAotCompatible>true</IsAotCompatible>

    <PublishAot>true</PublishAot>
</PropertyGroup>
```

This will enable more analyzers as per the trimming option that you turned on earlier. The warnings generated from the analyzers should not be ignored because they will most likely lead to runtime errors. This is a very important thing to consider when enabling trimming or AOT support because you will most likely be able to compile and publish your application; there is no guarantee that it will work though. I have confidence that following the testing chapters earlier, you will have created a suitable test suite to have the confidence that your application will behave at runtime.

Currently at the time of writing, Refit is not trim or NativeAOT compliant; therefore, despite showing how great it was to use back in Chapter 11, if you want to build an application that fully supports NativeAOT, you would need to remove the dependency on Refit. Let's proceed to fixing this to truly make the application support NativeAOT.

The first step here is to reintroduce the `WeatherForecastService` class from Chapter 11. Don't worry, I am not expecting you to keep jumping back; below is a detailed list of the items to change.

Remove Refit

The first step is to remove the Refit NuGet package from the project. I have opted to do this directly in the project file. Open the *WidgetBoard.csproj* file and delete the following entry:

```
<PackageReference Include="Refit.HttpClientFactory"
Version="7.2.1" />
```

Next we need to modify the *IWeatherForecastService.cs* file; you want to delete any code referring to Refit; thankfully the file is small so the result should look as follows:

```
namespace WidgetBoard.Communications;

public interface IWeatherForecastService
{
    Task<Forecast?> GetForecast(double latitude, double
    longitude, string apiKey);
}
```

Next we want to introduce the *WeatherForecastService.cs* file again. Add a new class file called WeatherForecastService and modify the contents to the following:

```
using System.Text.Json;

namespace WidgetBoard.Communications;

public class WeatherForecastService : IWeatherForecastService
```

```
{
    private readonly HttpClient httpClient;
    private const string ServerUrl = "https://api.
    openweathermap.org/data/2.5/weather?";

    public WeatherForecastService(HttpClient httpClient)
    {
        this.httpClient = httpClient;
    }

    public async Task<Forecast?> GetForecast(double latitude,
    double longitude, string apiKey)
    {
        var response = await httpClient
            .GetAsync($"{ServerUrl}lat={latitude}&lon=
            {longitude}&units=metric&appid={apiKey}")
            .ConfigureAwait(false);
        response.EnsureSuccessStatusCode();
        var stringContent = await response.Content
            .ReadAsStringAsync()
            .ConfigureAwait(false);
        var options = new JsonSerializerOptions
        {
            PropertyNameCaseInsensitive = true
        };
        return JsonSerializer.Deserialize<Forecast>
        (stringContent, options);
    }
}
```

Finally, we need to modify the registrations within *MauiProgram.cs*. Open that file up and make the following changes:

```
builder.Services
    .AddRefitClient<IWeatherForecastService>()
    .ConfigureHttpClient(c => c.BaseAddress = new Uri
    ("https://api.openweathermap.org/data/2.5"))
    .AddStandardResilienceHandler(static options =>
    {
        options.Retry = new HttpRetryStrategyOptions
        {
            BackoffType = DelayBackoffType.Exponential,
            MaxRetryAttempts = 3,
            UseJitter = true,
            Delay = TimeSpan.FromSeconds(2)
        };
    });
```

Wants to be changed to

```
builder.Services
    .AddHttpClient<WeatherForecastService>()
    .AddStandardResilienceHandler(static options =>
    {
        options.Retry = new HttpRetryStrategyOptions
        {
            BackoffType = DelayBackoffType.Exponential,
            MaxRetryAttempts = 3,
            UseJitter = true,
            Delay = TimeSpan.FromSeconds(2)
        };
    });
```

And also add this line into the `CreateMauiApp` method:

```
builder.Services.AddSingleton<IWeatherForecastService,
WeatherForecastService>();
```

Now that we have reintroduced the code, if you run a `dotnet publish` command, you will see that the tooling now reports a warning; let's take a look at how to fix that.

Warning IL3050

The warning that the tooling reports is as follows:

1>WeatherForecastService.cs(29,16): Warning IL3050 : Using member 'System.Text.Json.JsonSerializer.Deserialize<TValue>(String, JsonSerializerOptions)' which has 'RequiresDynamicCodeAttribute' can break functionality when AOT compiling. JSON serialization and deserialization might require types that cannot be statically analyzed and might need runtime code generation. Use System.Text.Json source generation for native AOT applications.

The reason a warning is reported is due to the fact that the deserialization code does not guarantee a hard reference to the properties of the `Forecast` class being deserialized. Now it may be say because our code might prevent the linker from removing the properties, but to be safe, we can make use of a source generator provided by System.Text.Json to make sure nothing will be trimmed away. In order to prevent this from happening, we can create a `ForecastContext` class and pass it into the `Deserialize` method. In fact, the last part of the warning message told us what to do just not how to do it. Let's take a look at this now; you should add the following class into your *Forecast.cs* file:

```
[JsonSerializable(typeof(Forecast))]
partial class ForecastContext : JsonSerializerContext
{
}
```

then you can change the call to `Deserialize` in the `GetForecast` method in the *WeatherForecastService.cs* file to match as follows (change in **bold**):

```
return JsonSerializer.Deserialize<Forecast>(stringContent,
ForecastContext.Default.Forecast);
```

This will tell System.Text.Json to use the new generator parsing context and make sure that no properties for the `Forecast` class are trimmed out. Running a `dotnet publish` command should also confirm that this warning has now been removed.

When Libraries Don't Support Trimming or AOT

There will be times when a library won't support trimming or AOT; in fact, both SQLite-net and LiteDB do not support them at the time of writing. We could take a similar approach that we did with Refit and find a suitable alternative that does support trimming – Microsoft.Data.Sqlite is one such example, which can be found at `https://www.nuget.org/packages/Microsoft.Data.Sqlite`. However, we will not be opting for this approach in favor of showing you how to exclude libraries from being trimmed.

In order to exclude a library from being trimmed, you can open your *WidgetBoard.csproj* file and add the following entry:

```
<ItemGroup>
    <TrimmerRootAssembly Include="LiteDB" />
</ItemGroup>
```

The above changes will exclude LiteDB from being trimmed from your application.

This now concludes the changes required to resolve warnings for trimming and NativeAOT support; let's take a look at the improvements it makes.

Results

I mentioned earlier that the use of trimming and NativeAOT can reduce application sizes. Table 16-1 shows the improvements.

Table 16-1. *Application sizes when using different trim modes and NativeAOT*

	Android	iOS	Mac Catalyst	Windows
Partial trim	31.8 MB	60.1 MB	121.5 MB	22.4MB
Full trim	29.7 MB	49.4 MB	102 MB	20.2 MB
NativeAOT	Not supported	22.2 MB	45.5 MB	15.5 MB

This concludes the section on optimizing your application. I would like to finish by saying that supporting trimming and NativeAOT is a great idea but I wouldn't pile too much pressure onto teams to fully support them from day one. There can be a lot of pain and complexities to deal with, especially when relying on a number of third-party packages which might not support it themselves.

Crashes/Analytics

Given that I have covered how things can go wrong, I would like to cover a way in which you can gain insight to when that happens. Each of the platform providers does offer a way to collect crash information and report it to you in order to make sure that you can prevent things like crashes from ruining the experience your applications provide.

There are frameworks/packages that aim to make this process easier by collecting and collating information from each platform into a centralized site. Further to this, you can enable the collection of analytic information to aid your understanding of how your users like to interact with your application and identify areas that you can improve upon.

In fact, a lot of the effort in my day job goes into finding ways to improve products. This only truly comes to light when you learn how your users interact with your applications. Capturing analytic information isn't the sole route I recommend taking. End user engagement can also be a fantastic thing to do if you have the opportunity. I would also like to highlight things like App Tracking Transparency by Apple and the Google equivalent as you want to make sure that when collecting analytic information, you are not passing on information that can be used to track your users, or you at least make them aware of it. Further to this, it is considered good practice to allow users to opt in to enable the collection of analytical information rather than just capturing it or making them opt out.

There are some companies that provide solutions for this already. They are fee based but do offer a free tier with fewer features.

Sentry

Sentry offers a .NET MAUI package that will make it easier to collect crash and analytical information. The website contains details on its usage and pricing: `https://sentry.io/for/dot-net/`.

Sentry also has the source code open sourced on GitHub and provides usage examples as well as assisting in understanding what the code does:

`https://github.com/getsentry/sentry-dotnet/tree/main/src/Sentry.Maui`

Obfuscation

It is a very safe assumption that if you are providing a compiled application to users' devices, any of the code in the application can be compromised, intellectual property (IP) can be stolen, or an attacker can learn about vulnerabilities in your application. If you really wish to retain your IP, then you likely want to keep it on a server-side component and have your application call it via a web API. That being said, there is still serious value in making use of tools that obfuscate the compiled code base to make it more difficult for an attacker to decipher what the application is doing.

Let's take a look at a simple class and how it will look when decompiled after obfuscation.

```
public class SomethingSecure
{
    private string PrivateSecret { get; } = "abc";
    internal string InternalSecret { get; } = "def";
    public string PublicSecret { get; } = "ghi";
}
```

The code decompiled using ILSpy without being obfuscated first looks as follows:

```
using System;
public class SomethingSecure
{
    private string PrivateSecret { get; } = "abc";
    internal string InternalSecret { get; } = "def";
    public string PublicSecret { get; } = "ghi";
}
```

If you run the original code through an obfuscation tool and then decompile the source, you will end up with something like the following:

```
// \u0008\u0002
using System;
[\u000f\u0002(1)]
[\u000e\u0002(0)]
public sealed class \u0008\u0002
```

```
{
    private readonly string m_\u0002 = \u0002\u0003.\
    u0002(-815072442);
    private readonly string m_\u0003 = \u0002\u0003.\
    u0002(-815072424);
    private readonly string m_\u0005 = \u0002\u0003.\
    u0002(-815072430);
    private string \u0002()
    {
        return this.m_\u0002;
    }
    internal string \u0003()
    {
        return this.m_\u0003;
    }
    public string \u0005()
    {
        return this.m_\u0005;
    }
}
```

It is clear from the above that it is much more difficult now to follow what this code is doing.

Obfuscation does not make it impossible for attackers to gain an understanding of what the code does. It does, however, make that task much more difficult. I would also add that you should consider any of the code within your client-based applications as insecure; ultimately if someone wanted to break into it to understand what is happening, they can and will. I would highly recommend making sure that any items such as passwords or web service tokens are stored in places like *SecureStorage* and not directly in code. For any important algorithms you implement, if you want these to remain your own intellectual property, you should keep these deployed to a server-side component which you can control access to.

Distributing Test Versions

There are a lot of different tools and websites that help you ship test builds out to people who can test your application. I have become most fond of using the deployment options provided by Apple and Google. The main reason I prefer to do it this way is that you do not need to change any of your deployment processes. You can continue to publish applications ready for releasing to the public via each store. In fact, these processes even upload the builds to the store portals. They simply allow you to release the application to a subset of users.

As is in keeping with this chapter, I won't walk you through each of these portals because the details can change from time to time. I refer you to the documentation provided by each platform provider and strongly urge you to investigate.

- Apple TestFlight, `https://testflight.apple.com`
- Google Play Internal Testing, `https://play.google.com/console/about/internal-testing/`

Summary

In this chapter, you have

- Explored the concepts of distributing your application
- Learned about continuous integration and continuous delivery to improve your development processes

- Learned about trimming, what it is, and how it can benefit/hinder you
- Learned about NativeAOT and the benefits it can bring
- Covered why it is important to collect analytical and crash information
- Explored why you may want to consider obfuscating your code
- Reached the end of our application-building journey together

CHAPTER 17

Conclusion

Abstract

Wow! If you made it this far, I want to thank you so much! I really hope that you have enjoyed reading this book as much as I enjoyed writing it. This book was designed to give you an insight into what .NET MAUI offers and how you can use it to build real-world applications. The sample we built together covers a lot of the key concepts. Of course I could have filled the book with hundreds more pages, adding in so many more widgets and features to the application. This application is a concept that is near and dear to my heart, so I can tell you that it will continue to evolve over time. I would love to hear where you decide to take it next, and I would love to see what you create next.

Looking at the Final Product

The application we just finished building together has been a pet project of mine for years, so thank you for helping me to finally reach this dream! Let's take a trip down memory lane to review what exactly we have built. Figure 17-1 shows my prototype sketch.

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0_17

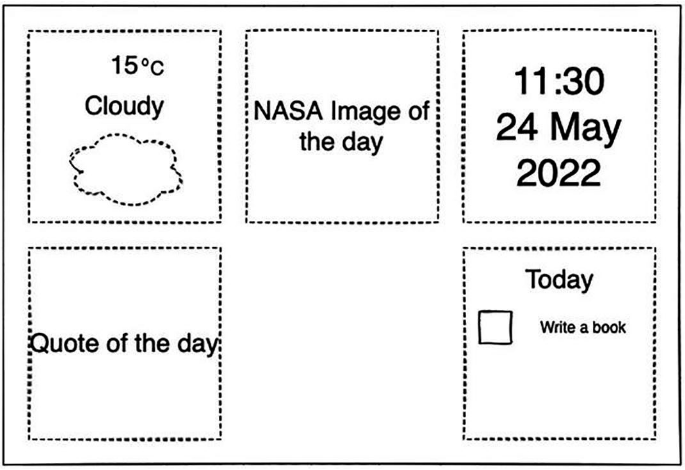

Figure 17-1. *Sketch prototype of the application*

The process of building this application has taken you through many different concepts including

- Creating a .NET MAUI application project
- Reviewing the possible architectural patterns you can use to build .NET MAUI applications
- Learning about the building blocks that make up your application's UI
- How you can further expand on the UI through styling
- How to make your application accessible
- How to create your own layout and utilize some cool features like `BindableLayout` to do a lot of the heavy lifting
- How to store data and the scenarios around where best to store each type of data

- How to access different types of remote data and the scenarios around when things go wrong
- How to customize your application on a per-platform basis
- How to test your application
- The concept of distributing your application
- How to further optimize your application with good practices and other features such as trimming and NativeAOT

All of the items above made for a really fun journey! And the end result is almost identical to my original plan. Figure 17-2 shows the final application with the widgets.

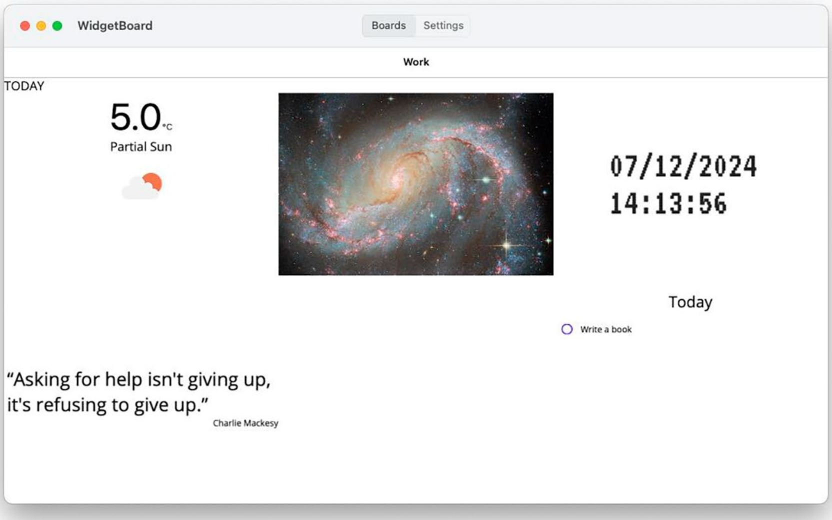

Figure 17-2. *The final application showing the widgets that we have added plus the results of some of the extra assignment sections*

Finally, in Figure 17-3, you can see the end result running on an old Kindle Fire HD device that I have sitting on my home office desk.

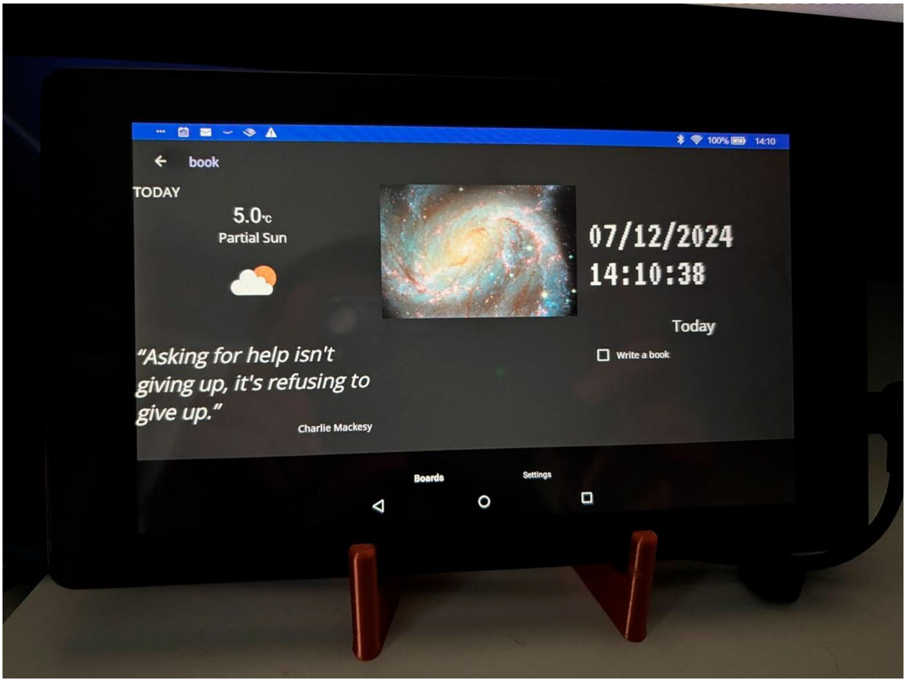

Figure 17-3. *The final application that I use on a daily basis*

Taking the Project Further

One main reason I really love this project is that I believe the possibilities of future widgets are wide open! I could provide a list as long as my arm on ideas that we could continue to achieve together. If I could, I would have fit them all in this book, but I would probably never have finished. Here is a short list of the things that I think we could achieve based on the knowledge that you have gathered in the book:

- Family planning calendar
- Image widget
  - Slideshow from device
  - Slideshow from external web service
- Shopping list
- Home assistant integration
- OctoPrint status widget
- Smart meter widget
- Social media follower count widget

I am repeating myself here, but I would really love to hear from you about your experience reading this book and where you have decided to take our application next.

Useful Resources

There are so many great places to find information on either building .NET MAUI applications or solving issues that may arise during that experience. The following list is a collection of websites that provide some really great content along with a few specific examples of content creators on those platforms.

StackOverflow

Stack Overflow (`https://stackoverflow.com`) is a question-and-answer site where you can seek assistance for issues that you encounter. Often someone else has already asked the question so you can find the answer you need. If you can't find a .NET MAUI-specific question/answer, it is worth also looking for Xamarin.Forms question/answers given that it is the predecessor to .NET MAUI.

GitHub

GitHub (https://github.com/dotnet/maui) is where the .NET MAUI repository is hosted and the framework is developed in the open. I strongly recommend keeping up to date with the discussions and issues on this repository.

YouTube

There are some really great content creators providing video tutorials on how to build .NET MAUI applications. Two great creators are in fact Microsoft employees; however, they build this content in their own free time, which I believe goes to show just how passionate they are about the framework.

Gerald Versluis

www.youtube.com/c/GeraldVersluis

James Montemagno

www.youtube.com/c/JamesMontemagno

Social Media

There is a whole host of social media options such as LinkedIn, Discord, Twitter, Bluesky, and Facebook. I urge you to find the platform that works best for you and start finding and following people that work on or with the technology.

Yet More Goodness

It is impossible to provide a curated list of all the great content creators or resources in printed form. It will instantly become outdated. In fact, by the time you have finished this book, you may well have become another name to add to this list! For that reason, here is a great resource that provides a curated list: `https://github.com/jfversluis/learn-dotnet-maui`.

Looking Forward

While .NET MAUI offers us a lot, there is still so much more that will evolve. I fully expect there to be some extensive work applied to improving the ability to test the user interfaces of .NET MAUI applications along with further enhancements in the usage of .NET MAUI Graphics, which has the potential to not only render applications identically across each platform (which is very similar to how Flutter works) but also to boost performance by moving away from the native controls that come with Android.

I feel the need to highlight the lack of sections here. In the first edition, I had points highlighting what I wanted to see in terms of better testing support. I was able to delete all of those items in this update because .NET MAUI provides support for all of it. That isn't to say it won't and shouldn't continue to evolve. I am sure concepts like automation testing will continue to become easier.

Thank you again for reading!

Index

A

AbsoluteLayout, 118–120, 225
Accessibility
applications
Android, 245
iOS, 245
macOS, 246
Windows, 246
definition, 227
dynamic text sizing, 240–245
principles, 228, 229
reasons, 228
resources
checklist, 247, 248
website, 248
screen reader support, 230–237
suitable contrast, 238–240
Adaptive icon, 111
AddAdditionalAppiumOption, 466
AddBoardCommand, 178, 179
AddScoped, 60
AddSingleton, 59
AddTransient, 60, 527
AddWidget method, 259, 310
Ahead-of-time (AOT), 11
Ahead-of-time compilation
AOT compilation, 531, 532
Refit NuGet package, 532–535
Warning IL3050, 535, 536
Analog clock widget
creation, 493–501
registering, 502
AnalogClockWidgetView, 494
modification, 494–496
AnalogClockWidgetViewModel
contents, 497–503
AnalogClockWidgetViewModel.cs file, 499
Android
Appium, 454
application, 509, 510
generation, 508
image sizes, 522
mappers, 413
ObservableCollection, 523, 524
platform-specific components, 397–399
Android App Bundle (AAB), 507, 509
Android driver, 454, 466–467
Android Package (APK), 507, 508
Android platform, 46, 47
accessibility, 245
app icons, 111
platform-specific lifecycle events, 67, 68

S. Lawrence, *Introducing .NET MAUI*, https://doi.org/10.1007/979-8-8688-1189-0

Android platform (*cont.*)
  screen reader, 230
  storage security, 340
Android Virtual Device (AVD), 466
Animations
  cancelling, 276
  chain, 276
  complex, 278–282
  concurrent, 276
  easings, 277, 278
  methods, 274
  prebuilt methods, 275
  triggers and, 283, 284
App data directory, 297
App icons, 50
  adding your own icon, 110, 111
  image resources, 109
  platform differences
    Android, 111
    iOS and macOS, 112
Appium, 452–456
Appium inspector, 457–459
AppiumLocalService, 461
Appium NuGet Package, 461
Appium Platform Drivers, 463–465
Appium Server, 453, 454, 461–463
Application building
  ahead-of-time compilation, 531–536
  Android, 507–510
  iOS, 511–515
  macOS, 506, 516–518
  .NET MAUI, 506
  trimming, 524–530
  Windows, 507, 517–519
Application lifecycle
  application states, 61, 62
  cross-platform lifecycle events, 65, 66
  handling lifecycle events, 63, 65
  lifecycle events, 62, 63
  platform-specific lifecycle events, 66–70
Application state lifecycle, 61, 62
ApplyQueryAttributes method, 221, 309, 329, 528
ApplyToDerivedTypes, 264
AppShell
  register, MAUI App builder, 163
  resolve, 163, 164
  setting BindingContext, 162
AppShellViewModel, 147, 155, 163, 302, 303
AppShell.xaml, 45, 142, 150, 157, 439
AppShell.xaml.cs, 45, 162, 303, 439
AppThemeBinding, 267
App.xaml, 45, 262, 439
App.xaml.cs, 45, 64, 163, 439
Assistive technologies, 228, 229, 240
Asynchronous operations
  ClockWidgetViewModel Mock, 436, 437
  creating your tests, 433–436
  creating your view tests, 437, 438

ILocationService Mock, 429, 430, 432
WeatherForecastService Mock, 432, 433
AutomationId
property, 472
AutomationProperties, 237
AutomationProperties.ExcludeWithChildren, 237
AutomationProperties.IsInAccessibleTree, 237
Automation testing, 549
add New Board Button, 473–476
Android Driver, 466, 467
Appium, 452, 453
Appium NuGet Package, 461
Appium Platform Drivers, 463, 465
Appium Server, 461–463
CollectionView, 478, 479
consistency, 459
create boards, 476–478
definition, 451, 452
installing Appium
Android, 454
iOS, 455
macOS, 454
Windows, 454
installing Appium Driver
macOS, 455
Windows, 456
installing Appium Inspector, 457–459
installing Node.js, 453
installing WinAppDriver, 456, 457
iOS Driver, 467
macOS Driver, 467, 468
parameterizing
Android configuration, 469
Command Line/Terminal, 471
iOS configuration, 469, 470
macOS configuration, 470
Visual Studio configuration, 471
Windows configuration, 471
Windows Driver, 468
writing, 472, 473

B

Base class
library (BCL), 4, 5, 525
BaseViewModel, 90–93
Behaviors
application, 288
approaches, 284
attaching, 287, 288
creation, 285, 286
BindableLayout, 197, 198
BindableLayout.ItemsSource, 197
BindableLayout.ItemTemplate, 197
BindableLayout.ItemTemplateSelector, 197
BindingContext, 87, 130, 131, 155, 162–163

Bindings
BindingContext, 130, 131
Command, 138–141
compiled, 141, 142
mode, 131, 133
MultiBinding, 135–138
Path, 131
remaining, 134, 135
source, 129, 133
target, 129
Board.cs, 106, 107, 314
BoardDetailsPage, 108, 142
BoardDetailsPageViewModel, 108, 109, 134, 300, 427
BoardLayout, 192, 252, 256
BoardLayout.xaml, 196–198
BoardLayout.xaml.cs, 198–203
creation, 195
BoardLayout.xaml, 196–198
BoardLayout.xaml.cs
adding remaining bits, 202, 203
ChildAdded event, 201, 202
coding, 198
ItemsSource property, 200
ItemTemplateSelector property, 200, 201
LayoutManager property, 198, 199
BoardListPage, 146, 148, 167, 170–171
BoardListPageViewModel, 148, 304, 305
BoardSearchHandler, 175, 176, 305–307
BoardWidgets, 307, 315, 317, 318, 325
Bootstrapping, 51, 55, 463

C

C#, 78, 86, 87
Cache directory, 296–297
Camera API, 13
Canvas, 485, 486, 493, 496
Chain animations, 276
ChangeCanExecute method, 140
CheckStatusAsync method, 386
ChildAdded event, 201, 202
Chosen approach
adding overlay view, 255, 256
showing overlay view, 259, 261
updating view model, 257–259
ClockWidget, 74, 217, 218
ClockWidgetView, 95–97, 218, 436–438
ClockWidgetViewModel, 93, 94, 135, 363
ClockWidgetViewModel Mock, 436, 437
C# markup, 85–88
Code generation libraries, 379
CollectionView, 159, 478, 479
Command, 138–141
Command line, 31
creating, 36, 37
CommunityToolkit.Maui, 381
CommunityToolkit.Maui.Markup, 87
Compiled bindings, 139, 141, 142, 257, 333, 525, 530

Compiler directives, 403, 404, 406, 416
Complex animations, 275, 278–282
Composition, 251, 285
Concurrent animations, 276, 283
Conditional statements, 409, 410
Constructor injection, 58, 130
ContainsKey method, 332
Content creators, 547–549
ContentPage, 77, 115, 116, 167, 178
Continuous delivery (CD), 504, 520, 541
Continuous integration (CI), 504, 520, 541
Controls, 6, 193
Converters, 371, 381
Crashes/analytics, 537, 538
CreateBoard method, 316, 323
CreateBoardWidget method, 316, 324
CreateMauiApp method, 55, 336, 391, 440
CurrentBoard, 157, 160, 161, 169

D

Database
- definition, 298
- entity relationship diagram, 298
- LiteDB, 319–326
- normalization, 298
- repository pattern, 299–311
- SQLite, 311–319
- types, 298

DataTrigger, 268–270
Dependency injection (DI), 5, 45, 55–58, 130, 148, 235, 358
Deserialization process, 354, 355
Design-time validation, 141
Device testing
- adding, 441, 442
- adding new project, 439
- adding Runner NuGet package, 439
- definition, 438
- remove extra entries, 441
- remove unnecessary contents, 439
- replace CreateMauiApp, 440
- replace WidgetBoard Project, 440
- running, 443, 444

dotnet, 33, 506
Draw method, 482–484, 486, 490
DrawingPath class, 490
Dynamic text sizing
- avoiding fixed sizes, 241–243
- font auto scaling, 244
- guideline, 241
- preferring minimum sizing, 244

E

Easings, 277, 278
Enable Developer Mode, 26, 27
EndInteraction event, 491
EnsureIndex method, 323
EnterActions, 270, 271
Entity relationship diagram, 298

EventTrigger, 268
ExitActions, 270, 271
Explicit styling, 264
eXtensible Application Markup Language (XAML), *see* XAML

F

File system
  abstraction, 296
  app data directory, 297
  cache directory, 296
FindById method, 325
FixedBoardPage, 146, 165, 166, 222
FixedBoardPageViewModel, 148, 218, 220, 222, 223, 307
FixedLayoutManager, 252
  building board layout, 209–211
  content, 204
  creation, 203
  NumberOfColumns property, 205, 206
  number of rows and columns, 205
  NumberOfRows property, 206, 207
  setting row/column position, 211, 212
  tap/click support, 207, 208
FlexLayout, 121, 122
Flyout
  application, 164–166
  board selection, 161
  collection of boards, 160, 161
  FlyoutContent, 159, 160
  FlyoutHeader, 158
  navigation to board selection, 162
  overview, 157
  register AppShell with MAUI App builder, 163
  resolve AppShell, 163, 164
  setting BindingContext your AppShell, 162, 163
FlyoutContent, 158–160
FlyoutHeader, 158
Font auto scaling, 244–245
Fonts, 50, 51
  application, 291
  clock widget rendering, 292
  configure, 291
  embed, 291
  purpose, 290
  usage, 291
Font scaling, 241, 244, 250
ForecastContext class, 535
Forecast.cs, 356

G

Generic Host Builder
  DI, 55–58
  registering dependencies, 59, 60
Geolocation API
  displaying permission errors, 396, 397
  ILocationService, 393–395
  registering LocationService, 393

- registering service, 391
- using service, 392, 393

GetForecast method, 358, 433, 536
Get method, 329, 331, 332
GitHub, 348, 349, 548
Google Play Store, 507, 509, 510
GoToAsync method, 179
GraphicsView, 482, 488, 496
Grid, 122–124, 197
GridUnitType.Star value, 210

H

Handlers
- abstract controls, 410
- BoardLayout class, 412
- customizing controls with mappers, 412–416
- .NET MAUI, 411
- renderers, 411

Hard-coded dependency approach, 55
HorizontalStackLayout, 125, 126
HttpClient backing field, 358

I

Images, 51–53
IBoardRepository interface, 311, 312, 320
ICanvas implementation, 482
IDispatcher implementation, 331
IDrawable interface, 482, 486, 496
ILayoutManager, 192, 194, 195, 204, 211
ILocationService, 393–395, 434
ILocationService Mock, 429, 430, 432
ImageSource, 361
Implicit styling, 263
Incidental text, 238
Inheritance, 262, 285
INotifyPropertyChanged interface, 79, 427, 428
Intermediate language (IL), 11
Inversion of Control (IoC), 57
iOS, 1, 11, 20, 47, 68–69, 112, 230, 245–246, 341, 399–401, 511
iOS Driver, 455, 467–468
iOS platform, 47
- accessibility, 245
- additional resources, 515
- app icons, 112
- Appium, 455
- arguments, 514, 515
- capabilities, 513
- certificate, 511, 512
- entitlements, 513
- identifier, 512
- mappers, 413
- platform-specific components, 399, 401
- platform-specific lifecycle events, 68, 69
- provisioning profiles, 513, 514
- screen reader, 230
- storage security, 341

IQueryAttributable interface, 155, 156, 528
IsEqualToStateConverter, 374, 375
ItemsSource property, 174, 175, 197, 200
ItemTemplateSelector property, 200, 201
IValueConverter interface, 372
IWeatherForecastService, 356, 363, 380, 381, 435
IWidgetView, 94, 95, 202
IWidgetViewModel, 89, 90

J

JetBrains Rider, 16
Just-in-time (JIT), 11

K

Keychain API, 341

L

Label.FormattedText, 361
Large-scale text, 238
LayoutManager property, 198, 199
Layouts, 194
- AbsoluteLayout, 118–120
- creation, 213
- FlexLayout, 121, 122
- Grid, 122–124
- HorizontalStackLayout, 125, 126
- updating FixedBoardPage ViewModel, 220, 222
- VerticalStackLayout, 126–129
- widgets
  - ClockWidget, 217, 218
  - creation, 215–217
  - MauiAppBuilder, 217
  - purposes, 213
  - registration and view models, 214, 215
- WidgetTemplateSelector, 218–220

Lifecycle events, 62, 63
- cross-platform mappings, 65, 66
- handling, 63, 65

LINQ-based expressions, 317, 324
Linux development, 19
LiteDB
- connecting, 322
- creating tables, 323
- defined, 319
- deleting database, 325
- inserting database, 323
- installing, 320
- mapping, 322
- reading database, 324
- reading single entity database, 325
- updating an entity in database, 326
- usage, 320, 321

LiteDBBoardRepository, 320
LoadBoards method, 303, 304, 330

LoadWeatherForecast method, 394
Local data
  application settings (preferences), 327–336
  back up and widget, 342
  database (*see* Database)
  definition, 295, 296
  filesystem, 296, 297
  options, 296
  setting page, 343
  storage security, 336–341
  viewing result, 341
LocationAlways, 387
LocationService, 393–395, 403
LocationWhenInUse, 386–388
Logotypes, 238

M

MacCatalyst, 48
  platform-specific lifecycle events, 68, 69
macOS
  accessibility, 246
  app icons, 112
  clock widget application, 99
  MauiReactor, 82
  .NET MAUI workload, 32
  Remote Access, 22–25
  remote login options, 25
  sharing options, 24
  system settings, 23
  Visual Studio, 27–30
  Xcode, 20–22
macOS Driver, 467, 468
macOS platform, 506
  Appium, 454
  Appium Driver, 455
  application, 516, 517
  mappers, 413
  platform-specific components, 399, 401
  resources, 517
  storage security, 341
MainPage.xaml, 45, 97
MainPage.xaml.cs, 45, 98
MainThread.InvokeOnMainThreadAsync method, 388
Mappers
  on Android, 413
  applying globally, 414, 415
  applying single instances, 416
  applying single page, 415, 416
  handler architecture, 412
  on iOS/Mac catalyst, 413
  on Windows, 414
MauiAppBuilder, 217, 220, 502
MauiAsset, 53
MauiProgram.cs, 45, 108, 109, 146–149, 220, 291, 313
MauiReactor
  definition, 81
  macOS, 82
  Windows, 82
MauiSplashScreen, 54
Measuring stick, 73, 74
Media playback, 14, 15
MockClockWidgetViewModel, 438

Mockup of board, 191, 192
Modal, 252, 253
Model View Update (MVU)
  benefits, 81
  implementation, 83–85
  MauiReactor, 81, 82
  overview, 80, 81
  parts, 80
  project format, 82, 83
Model View ViewModel (MVVM)
  adding views, 94–97
  benefits, 76
  choosing, 88
  enhancements
    frameworks, 99, 100
    magic, 100–103
  model, 76
  overview, 75, 76
  view
    C# (code-behind), 78
    ContentPage/ContentView, 77
    definition, 77
    XAML, 77
  viewing your widget
    application, 98, 99
    modifying MainPage.xaml, 97
    modifying MainPage.xaml.cs, 98
  ViewModels, 78–80, 89–94
Mono runtime, 4, 5, 531
MSTest, 421
MultiBinding, 135–138
MultiTrigger, 269

N

Navigation
  backward, 154
  flyout, board selection, 162
  IQueryAttributable, 155, 156
  passing data, 154, 155
  performing, 153
  QueryProperty, 156, 157
  registering pages, 151–153
  Tabs, board selection, 170
.NET 9.0, 35, 142, 521, 525
.NET Framework selection dialog, 36
.NET Hot Reload, 10
.NET MAUI Blazor Hybrid, 9, 13
.NET MAUI Community Toolkit, 15, 87, 153, 254
.NET MAUI ContentView (XAML), 95, 196, 359, 487, 494
.NET MAUI essentials
  concepts, 386
  configuring platform-specific components, 397
  geolocation API, 391–397
  handlers, 410–416
  overriding, platform-specific UI, 407–416
  permissions, 386–391
  platform-specific API access, 403–407
.NET MAUI Graphics, 481
  Draw method, 482, 483
  paths, 483, 484

pixel-perfect graphics, 482
surface updation, 482
.NET Multi-platform App UI (.NET MAUI)
application tasks, 6
building applications, 3, 15–17
Button control, 6
code sharing, 2, 8, 9
commercial offerings, 11, 12
community, 9
competition, 6, 7
compiling and running, 4
components, 2
concepts, 544, 545
definition, 1
developer freedom, 9
development cycle
.NET Hot Reload, 10
XAML Hot Reload, 10
full breakdown, 5
home office desk, 546
interacting APIs, 2
knowledge, 546
limitations, 12–14
performance, 10, 11
platform implementations, 13
platform-specific APIs, 3
platform support, 2, 7, 8
project structure, 43–54
resources, 547
selecting project type, 34
sketch prototype, 543, 544
unit testing (*see* Unit testing)
user interfaces, 549
widgets, 545
workload, 31, 32
Xamarin, 5
Network connectivity, 349
Network resilience handling, 376–378
Newtonsoft, 353
NumberOfColumns property, 205, 206
NumberOfRows property, 206, 207
NUnit, 421, 459, 463

O

Obfuscation, 538–541
Object relational mapping (ORM), 326
ObservableCollection, 160, 523, 524
OnAddBoard method, 179, 181, 184
OnAppearing method, 303, 304
OnBindingContextChanged method, 199
OnItemSelected method, 174, 175
OnNavigatedTo method, 303–305
OnPlatform, 407
OnPlatform Markup Extension, 408, 409
OnPropertyChanged method, 91, 92
OnQueryChanged method, 173, 174, 306

Open Weather API
connecting, 356–359
creating account, 350
creating API key, 351
creating your models, 354–356
examining data, 351–353
registering your widget, 365, 366
running and showing, 367, 368
System.Text.Json, 353, 354
testing your widget, 366, 368
WeatherWidgetView, 359–362
WeatherWidgetViewModel, 362–365
OpenWeatherApiToken, 338, 339
Open Web Application Security Project (OWASP), 349
Operating systems, 19, 20, 63, 339
Overlaying-a-view approach, 254
Overriding, platform-specific UI
conditional statements, 409, 410
OnPlatform, 407
OnPlatform Markup Extension, 408, 409

P

Pair to Mac screen, 29, 30
Path element, 131
Permissions
checking status, 386, 387
handling applications, 389, 391
requesting, 388, 389
Permissions.RequestAsync method, 388
Placeholder, 176, 188, 192–194, 197
PlatformLocationService, 405, 438
PlatformLocationServiceTests, 441
Platforms/folder, 44
Android, 46, 47
iOS, 47
MacCatalyst, 48
Tizen, 48
Windows, 49
Platform-specific API access
compiler directives, 403, 404
folders, 405–407
Platform-specific components
Android, 397–399
exceptions and error messages, 397
iOS/Mac, 399, 401
Windows, 402
Platform-specific lifecycle events
Android, 67, 68
iOS and MacCatalyst, 68, 69
Windows, 69, 70
Popups, 254
Preferences
checking key, 332
defined, 327
displaying, 332–336
getting value, 329–331
implementation, 327
removing, 332
setting a value, 328, 329
storage, 327, 328
preferences.Get method, 331

Prerequisites, 74, 75
 Shell
  pages, 145–147
 user interface essentials
  models, 106, 107
  pages, 107, 108
  ViewModels, 108, 109
Project structure, .NET MAUI
 AppShell.xaml and AppShell.xaml.cs, 45
 App.xaml, 45
 App.xaml.cs, 45
 MainPage.xaml and MainPage.xaml.cs, 45
 MauiProgram.cs, 45
 Platforms/ folder, 44, 46–49
 Resources/ folder, 45, 49–54
 single project, 43
Property changed method, 206

Q

QueryProperty, 156, 157, 528

R

Raw files, 53
Refit NuGet Package, 379–381, 532–535
RegisterWidget method, 214
Relative luminance, 238
Remote access, 22–25
Remote data, 295
 definition, 347
 failures, 349
 loading times, 348
 security, 349, 350
 web services (*see* Web services)
Repository pattern
 AppShellViewModel, 302, 303
 BoardListPageViewModel, 304, 305
 BoardSearchHandler, 305–307
 coding, 299
 creating board, 300, 301
 defined, 299
 loading board, 307–311
ResourceDictionary, 113
Resources/ folder, 45
 AppIcon, 50
 fonts, 50, 51
 images, 51–53
 raw files, 53
 splash, 54
 styles, 54
RouteNames, 152, 153
Rubber band animation, 280–282
RunAfterAllTests, 465
RunBeforeAnyTests, 465
Runsettings, 465, 467, 469, 470

S

SaveCommand model, 138
SaveWidget method, 310
Scalable Vector Graphics (SVG), 52
ScaleTo animation, 278, 279
Scoped registration, 60

Screen reader support
 on Android, 230
 AutomationProperties, 237
 AutomationProperties.
  ExcludeWithChildren, 237
 AutomationProperties.
  IsInAccessibleTree, 237
 concept, 230
 on iOS, 230
 navigation, 231–235
 SemanticScreenReader, 235, 237
SearchHandler
 adding data, 172, 173
 application, 176, 177
 BoardSearchHandler, 175, 176
 defined, 172
 implementation, 172
 inherit, 173
 OnItemSelected method,
  174, 175
 OnQueryChanged method,
  173, 174
Security
 remote data, 349, 350
SemanticProperties class
 code setting, 234, 235
 defined, 231
 SemanticProperties.
  Description, 231–233
 SemanticProperties.
  HeadingLevel, 233
 SemanticProperties.Hint, 233
SemanticProperties.Description
  property, 231–233
SemanticProperties.HeadingLevel
  property, 233
SemanticProperties.Hint
  property, 233
SemanticScreenReader, 235, 237
Sentry, 538
ServiceProvider, 59, 215
Set method, 328, 329
SetProperty method, 92, 221
Setter, 264
SettingsPage, 146
SettingsPageViewModel, 149, 334
Shell
 definition, 149
 flyout, 157–166
 navigation, 151–157
 prerequisites, 145–149
 search, 172
 ShellContent, 150
 tabs, 166–171
 ToolbarItems, 177–187
 ViewModels, 147–149
ShellContent, 150
ShouldShowRationale, 387
ShowOverlayTriggerAction, 272,
  273, 275
Singleton registration, 59
Sketch widget, 492
 SketchWidgetViewModel,
  485, 486
SketchWidgetViewModel, 485, 486
SketchWidgetView.xaml, 487, 488
SketchWidgetView.xaml.cs,
  488–492

Snapshot testing
  application, 445
  golden master, 444
  Verify.Xunit NuGet Package, 446
  WidgetBoard and WidgetBoard.
    Tests, 446, 448, 449
Social media, 547, 548
Splash screen, 54, 112, 113
SQLite
  connecting, 313, 314
  creating tables, 316
  defined, 311
  deleting database, 319
  inserting database, 316
  installing, 311
  mapping information, 314, 315
  reading database, 317
  reading single entity from
    database, 317, 318
  updating database, 319
  usage, 312, 313
SqliteBoardRepository, 312, 316
Sqlite-net-pcl, 311, 318
sqlite-net-sqlcipher, 326
Stack Overflow, 547
Startup tracing, 522
StateContainer, 381
Stockholm syndrome, 87
Storage security
  coding, 336
  constructor, 337
  CreateMauiApp method, 336
  fields and properties, 337
  platform-specific APIs
    Android, 340
    iOS and macOS, 341
    Windows, 341
  reading value, 339
  removing value, 339
  storing value, 338
Styles, 54, 113, 262
Styling
  advantages, 261–263
  AppThemeBinding, 267
  creation, 265, 267
  default
    ApplyToDerivedTypes, 264
    setter, 264
    TargetType, 263
  overlay, 266
Suitable contrast, 238–240
System.Text.Json, 353, 354, 535

T

Tabs
  application, 171
  BoardDetailsPage, 166
  board selection, 169
  collection of boards, 168
  ContentPage element, 167
  navigation to board selection, 170
  setting BindingContext uour
    BoardListPage, 170, 171
  SettingsPage, 167
Target net9.0, 423, 424
TargetType, 263–264
Temperature property mapping, 355

TestCase attribute, 476
Test draw, 492–493, 502, 503
Test versions, 541
Tizen platform, 48
ToolbarItems
AddBoardCommand, 179
application, 184–187
concepts, 178
adding ContentPage, 178
ContentPage
returning result, 183, 184
waiting for result, 181–183
PresentationMode property
adding cancel button, 180, 181
adding ContentPage element, 180
overview, 180
showing page and wait for result, 181
Transient registration, 60
TriggerAction, 271–273
Triggers
and animations, 283, 284
DataTrigger, 269, 270
defined, 268
EnterActions and ExitActions, 270, 271
TriggerAction, 271–273
types, 268
Trimming, 524
analogy, 525
command, 526
libraries, 536
third-party packages, 526
Warning IL2026, 527–530
Warning IL2087, 526
XAML compiler, 525
Troubleshooting installation issues, 31, 32
TypeConverters, 361

U

UIAutomator2, 454
UI thread, 331, 388
Unit testing
adding new project dialog, 422
adding project to your solution, 422, 423
asynchronous operation (*see* Asynchronous operations)
definition, 419, 420
framework, 421
MAUI dependencies, 425, 426
MSTest, 421
NUnit, 421
Reference Manager, 425
reference your project, 424
Target net9.0, 423, 424
view models, 426–428
xUnit, 420
User interaction, 85, 138, 234, 486, 487
User interface essentials
app icons, 109–112
application, 143
bindings, 129–142
BoardDetailsPage, 142

layouts, 118–129
prerequisites, 105–109
splash screen, 112, 113
XAML, 113–118
User interface renders, 13, 14

V

ValidForecastResultsInSuccessful Load test, 445
VerifyTests, 445
VerticalStackLayout, 115, 116, 126–129
View-based file, 113
ViewModels, 78–80
adding IWidgetViewModel, 90
MVVM-based architecture, 89–94
Shell, 147–149
user interface essentials, 108, 109
Views, 77, 78, 94–97
Vision impairment, 228
VisualElement, 274, 275, 277, 280
Visual Studio, 15, 16
automation testing, 471
building and running, 37–39
build target selection drop-down, 38
configure your project dialog, 35
creating, 33–36
macOS, 27–30
toolbar with Pair to Mac buttons, 30
Windows, 25, 26
Visual Studio Code, 16
Visual Studio Installer, 31

W

Warning IL2026, 527–530
Warning IL2087, 526, 527
Warning IL3050, 535, 536
WeatherForecastService class, 532
WeatherForecastService Mock, 432, 433
WeatherWidgetView, 359–362
WeatherWidgetViewModel, 362–365, 428, 445
Web Assembly (WASM) support, 13
Web Content Accessibility Guidelines (WCAG), 228, 247
Web services
adding Refit NuGet Package, 379–381
code generation libraries, 379
converting state to UI, 371, 372
displaying error state, 375, 376
displaying loaded state, 374
displaying loading state, 372
loading code, 369
network resilience handling, 376, 378
Open Weather API, 350–368
prebuilt libraries, 379
visual feedback, 369
WidgetBoard, 38, 40, 41
WidgetBoard.AutomationTests project, 472
WidgetBoard Project, 74, 82, 172, 285, 440
WidgetBoard.SnapshotTests, 445

WidgetBoard solution, 365, 377, 379, 460
WidgetFactory, 213, 258
Widgets
  application, 545
  chosen approach, 254–261
  ClockWidget, 217, 218
  creation, 215–217
  MauiAppBuilder, 217
  overlaying view, 253
  purposes, 213
  registration and view models, 214, 215
  showing modal page, 252, 253
  showing popup, 254
WidgetTemplateSelector, 218–221
WinAppDriver, 456, 457
Windows, 507, 517, 518
  accessibility, 246
  Appium, 454
  Appium Driver, 456
  application, 518, 519
  enable Developer Mode, 26, 27
  firewall request, 28
  mappers, 414
  MauiReactor, 82
  .NET MAUI workload, 32
  platform, 49
  platform-specific components, 402
  platform-specific lifecycle events, 69, 70
  storage security, 341
  Visual Studio, 25, 26
Windows Driver, 456, 468
Windows Software Development Kit (SDK), 246
Windows UI Library (WinUI) 3, 11

X

Xamarin Essentials, 6, 296
Xamarin.Forms, 5, 44, 58
Xamarin framework, 5
XAML, 77
  building your first page, 116–118
  definition, 113
  dissecting, 114–116
  OnPlatform markup extension, 408
  types
    ResourceDictionary, 113
    view-based file, 113
XAML Hot Reload, 10
XAML *vs.* C# Markup, 85–88
Xcode, 16, 20–22, 511
XCUITest, 455
xUnit, 420, 421

Y

YouTube, 548

Z

Z-index, 256