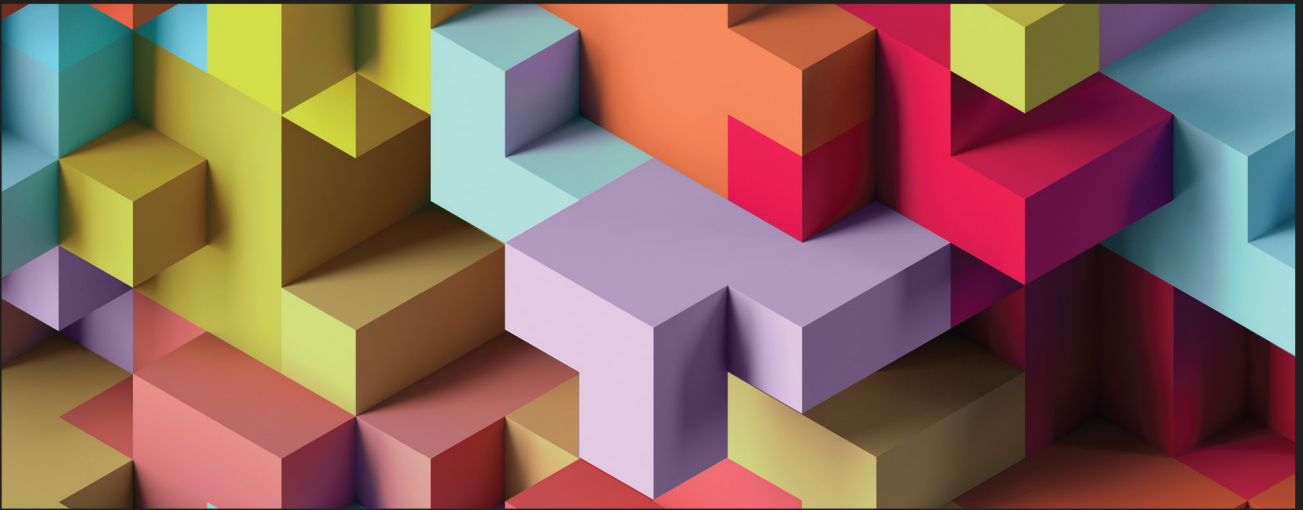




1ST EDITION

.NET MAUI Cross-Platform Application Development

Leverage a first-class cross-platform UI framework
to build native apps on multiple platforms

ROGER YE

.NET MAUI Cross-Platform Application Development

Leverage a first-class cross-platform UI framework to build native apps on multiple platforms

Roger Ye



BIRMINGHAM—MUMBAI

.NET MAUI Cross-Platform Application Development

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Nitin Nainani

Senior Editor: Keagan Carneiro

Senior Content Development Editor: Debolina Acharyya

Technical Editor: Simran Udasi

Copy Editor: Safis Editing

Project Coordinator: Sonam Pandey

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Aparna Bhagat

Marketing Coordinator: Nivedita Pandey

First published: February 2023

Production reference: 1050123

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-922-5

www.packt.com

*To my family: my wife Bo Quan and my daughter Yuxin Ye,
and the memory of my father and my mother.*

– Roger Ye

Contributors

About the author

Roger Ye is a software engineering manager who has worked in the software industry for many years.

Roger started his career as a software engineer in embedded system development at companies such as Motorola, Emerson, and Intersil. During this period, he wrote two books about embedded system programming, *Embedded Programming with Android* and *Android System Programming*.

In 2013, Roger joined McAfee as a software engineering manager. He recently started working at EPAM Systems, moving from system programming to application programming. His first book on application programming is on .NET MAUI.

I want to thank the team at Packt, who worked very hard with me to keep to schedule.

About the reviewers

Glenn Stephens is a software developer and product designer, implementing solutions for the mobile-, desktop-, and cloud-centered worlds we all live in. Glenn has worked in many different roles, such as managing director, chief executive officer, solution architect, software development manager, and programmer; has worked in fields spanning high-end security, e-health, education, and finance; and has won several awards along the way. Glenn has a bachelor's degree in computer science, an MBA with a specialization in e-business, and a Graduate Certificate in Arts in theatre and performance. He considers himself a lifelong learner.

An author, speaker, and product builder, he has been writing code since the 80s, with the musical taste to match. When he's not working, he enjoys playing the piano, reading, and spending time with his children.

Rohit Vinod Kelkar is an artist, a techie, and an experienced mobile application developer who is enthusiastic about cross-platform mobile application technologies. He has expertise in technologies such as Xamarin, Flutter, native iOS, and .NET as a full stack developer. He was an early adopter of .NET MAUI and works with a community of developers to share updates and blog about the technology. He also shares an interest in consulting and helping products in their initial stages.

Siddharth Singh has over 12 years of software development experience, having worked on a variety of platforms such as Windows, web, and mobile applications. Having worked at Expedia, Adobe, and Salesforce, his experience ranges from COM-/ASP-based legacy apps to React/ASP.NET Core modern applications. While working on Xamarin apps, he developed a passion for the Microsoft cross-platform framework.

He currently works at Microsoft as a senior developer for Azure Data Factory, writing data integration applications. When not at work, you can find him reading about theoretical computer science and artificial general intelligence.

Table of Contents

Preface	xiii
---------	------

Part 1: Exploring .NET MAUI

1

Getting Started with .NET MAUI	3
An overview of cross-platform technologies	3
Native applications	4
Web applications	4
Backend services	4
Cross-platform technologies	4
A comparison of .NET, Java, and JavaScript	5
Exploring the .NET landscape	7
.NET Framework	7
Mono	7
.NET Core	8
.NET Standard and portable class libraries	8
Using Xamarin for mobile development	9
Xamarin.Forms	10
Xamarin.Essentials	10
Moving to .NET MAUI	12
.NET MAUI Blazor apps	14
Choosing XAML versus Razor in .NET MAUI	15
Development environment setup	15
Installing .NET MAUI on Windows	16
Installing .NET MAUI on macOS	17
What you will learn in this book	19
The app that we will build in this book	19
Summary	20
Further reading	20

2

Building Our First .NET MAUI App	21
Technical requirements	21
Managing the source code in this book	21
Setting up a new .NET MAUI project	23
Creating a new project using Visual Studio	23

Creating a new project using the dotnet command	27	Building and debugging	43
App startup and lifecycle	27	Windows	44
Lifecycle management	29	Android	44
Configuring the resources	36	iOS and macOS	45
App icon	37	Scaffolding a Model-View-ViewModel project	47
Splash screen	37	Migrating and reusing a Shell template from Xamarin.Forms	49
Setting custom font icons	38	Visual Studio project template	54
		Summary	56

3

User Interface Design with XAML **57**

Technical requirements	58	Stacked	69
Creating a XAML page	58	Controls in .NET MAUI	70
XAML syntax	60	Layouts in .NET MAUI	71
Element	60	Navigation in the master-detail UI design	74
Attribute	62	Supporting multiple languages – localization	81
XML namespaces and XAML namespaces	62	Creating a .resx file	81
XAML markup extensions	66	Localizing text	83
Master-detail UI design	67	Summary	85
Side-by-side	69	Further reading	86

4

Exploring MVVM and Data Binding **87**

Technical requirements	87	Improving the data model and service	98
Understanding MVVM and MVC	88	KPCLib	99
MVVM in PassXYZ.Vault	89	PassXYZLib	102
Data binding	90	Updating the model	103
Binding mode	94	Updating the service	103
Changing notifications in viewmodels	96		

Binding to collections	104	Further reading	111
Summary	111		

5

Navigation using .NET MAUI Shell and NavigationPage 113

Technical requirements	113	Tabs	122
Implementing navigation	114	Shell navigation	125
INavigation interface and NavigationPage	114	Improving our model	133
Using the navigation stack	115	Understanding the data model and its services	134
Manipulating the navigation stack	116	Improving the login process	137
Using Shell	117	The Command interface	140
Flyout	119	Summary	143

6

Introducing Dependency Injection and Platform-Specific Services 145

Technical requirements	145	Using built-in MS.DI DI service	151
A quick review of design principles	146	Connecting to the database	160
Exploring types of design principles	146	Initializing the database	162
Using design principles	147	Performing CRUD operations	163
Using DI	149	Summary	172
Dependency Service	149	Further reading	172

Part 2: Implementing .NET MAUI Blazor

7

Introducing Blazor Hybrid App Development 175

Technical requirements	175	Creating a new .NET MAUI Blazor project	182
What is Blazor?	176	Generating a .NET MAUI Blazor project with the dotnet command line	183
Learning about Blazor Server	176		
Understanding Blazor Wasm	177		
Exploring Blazor Hybrid	179		

Creating a .NET MAUI Blazor project using Visual Studio on Windows	184	Directives	193
Running the new project	185	Directive attributes	193
The startup code of the .NET MAUI Blazor app	187	Creating a Razor component	194
Migrating to a .NET MAUI Blazor app	190	Redesigning the login page using a Razor component	194
Understanding Razor syntax	191	The Model-View-ViewModel (MVVM) pattern in Blazor	201
Code blocks in Razor	191	Dependency injection in Blazor	203
Implicit Razor expressions	191	CSS isolation	204
Explicit Razor expressions	192	Summary	206
Expression encoding	192		

8

Understanding the Blazor Layout and Routing 207

Technical requirements	207	Nesting layouts	220
Understanding client-side routing	208	Implementing navigation elements	220
Setup of BlazorWebView	208	Implementing a list view	222
Setup of Router	209	Adding a new item and navigating back	228
Defining routes	210	Summary	230
Using Blazor layout components	214		
Applying a layout to a component	218		

9

Implementing Blazor Components 233

Technical requirements	233	Data binding	243
Understanding Razor components	234	Component parameters	244
Inheritance	236	Nested components	246
Creating a Razor class library	237	Two-way data binding	250
Using static assets in the Razor class library	239	Communicating with cascading values and parameters	253
Creating reusable Razor components	239		
Creating a base modal dialog component	241		

Understanding the component lifecycle	255	OnAfterRender and OnAfterRenderAsync	258
SetParametersAsync	256	Implementing CRUD operations	262
OnInitialized and OnInitializedAsync	257	CRUD operations of items	262
OnParametersSet and OnParametersSetAsync	257	CRUD operations of fields	266
ShouldRender	258	Summary	271

10

Advanced Topics in Creating Razor Components 273

Technical requirements	273	Using the ListView component	284
Creating more Razor components	274	Built-in components and validation	285
Creating the Navbar component	274	Using built-in components	286
Creating a Dropdown component for the context menu	277	Using the EditForm component	287
		Creating an EditFormDialog component	288
Using templated components	281	Summary	299
Creating a ListView component	282	Further reading	300

Part 3: Testing and Deployment

11

Developing Unit Tests 303

Technical requirements	303	Creating a bUnit test case	319
Unit testing in .NET	304	Creating test cases in Razor files	320
Setting up the unit test project	305	Using the RenderFragment delegate	323
Creating test cases to test the IDataStore interface	307	Testing Razor pages	327
Sharing context between tests	309	Summary	332
		Further reading	333
Razor component testing using bUnit	317		
Changing project configuration for bUnit	318		

12

Deploying and Publishing in App Stores		335
Technical requirements	335	Publishing to the Google Play Store 343
Preparing application packages for publishing	336	Publishing to Apple’s App Store 347
What to prepare for publishing	336	GitHub Actions 352
Publishing to Microsoft Store	337	Understanding GitHub Actions 353
		Summary 362
Index		363
Other Books You May Enjoy		374

Preface

In 2017, when we hit performance issues in one of our projects using Cordova, I started to look for an alternative cross platform programming solution. Xamarin was one of the solutions that I investigated at that time. From then until now, I have spent many years on the development of Xamarin and its descendant .NET MAUI. It's one of the technologies that I love.

Even though we have more cross platform programming options today, such as Flutter or React Native, .NET MAUI has some particularly unique features that we may consider when we are looking for a cross-platform solution.

.NET MAUI uses a single project structure, which is a major improvement compared to Xamarin.Forms. With a single project, we can see the improvement in the following areas:

- **Better debug and test experience** – We can select and debug multiple targets in one project. We don't have switch to different projects to debug or test different targets.
- **Sharing resources** – In Xamarin, we have to manage resources in each platform individually. With the improvement of .NET MAUI, we can share most resources cross-platform, such as fonts, images, icons, and so on.
- **Simplified configuration** – We can use a single app manifest most of time, so we don't need to manage platform configuration files separately, such as `AndroidManifest.xml`, `Info.plist`, or `Package.appxmanifest`.

In Flutter or React Native, you can use the Flutter plugin or React Native module to access native device features. To use plugins or native modules, you have to rely on the developer community, or you have to develop your own. These interfaces are designed by developers, so they are not standardized. In .NET MAUI, Microsoft has done the job of standardizing APIs for the most frequently used native device features as part of the .NET MAUI release.

In .NET MAUI, we not only can develop apps using the traditional XAML-based UI, but we also can develop Blazor-based UIs as Blazor Hybrid apps. This opens a door for a higher-level reuse of source code. If you are working on a project that includes a web and mobile app, you can even share the **user interface (UI)** design and source code between the web and mobile app.

Since .NET MAUI is part of the .NET platform release now, we can always use the latest .NET platform and C# language features with each release of .NET. We can use advanced features, such as .NET generic hosting, dependency injection, or the MVVM Toolkit from the latest .NET release.

In this book, I will share my journey in .NET MAUI development with you using the open source app that I have developed. Both .NET MAUI and .NET platform features will be covered in this book.

Who this book is for

This book is for frontend developers or native app developers who want to explore cross platform programming technology. This book assumes the audiences have C# programming knowledge or knowledge of any object-oriented programming language similar to C#.

What this book covers

Chapter 1, Getting Started with .NET MAUI, provides an introduction to cross-platform technologies. As part of the introduction, .NET MAUI is compared with other cross-platform technologies. The .NET MAUI development environment setup is also covered in this chapter. You will be given an overview of cross-platform technologies that can help you to make the choice for your own project.

Chapter 2, Building Our First .NET MAUI App, is about setting up the new project for the development work in this book. The .NET MAUI project structure and application life cycle will be discussed as well. You will learn how to create a new project and some basic debugging skills for a .NET MAUI app.

Chapter 3, User Interface Design with XAML, covers the UI design using XAML. Basic knowledge of the XAML and .NET MAUI UI elements will be discussed. By the end of this chapter, you will be able to work on your own UI design.

Chapter 4, MVVM and Data Binding, explains some key topics in .NET MAUI app development, including the MVVM pattern and data binding. We will start with the theory first and then apply what we have learned to the development work of the password management app. You will learn how to use data binding and apply it to the MVVM pattern.

Chapter 5, Introducing Shell and Navigation, introduces Shell and navigation in .NET MAUI. We use Shell to build the skeleton and navigation hierarchy of our app. You will learn about the usage of navigation stack and the Shell elements, which can help you to create your application layout and navigation hierarchy.

Chapter 6, Dependency Injection and Refining Design, discusses design principles and provides an overview of SOLID design principles. After that, we explain the usage of dependency injection in .NET MAUI. We also apply it in our app development. In this chapter, you will get an overview of the SOLID design principles and see a deep dive into dependency injection.

Chapter 7, Introducing .NET MAUI Blazor, takes .NET MAUI Blazor application development as its central topic. We will demonstrate how to create a new Blazor Hybrid app and teach you how to convert a .NET MAUI XAML app into a .NET MAUI Blazor Hybrid app. You will learn about the basic environment setup and Razor syntax in this chapter.

Chapter 8, Understanding Blazor Layout and Routing, explores the layout and routing of Blazor Hybrid apps. We will learn about the router setup and layout components. You will learn how to create a layout and set up routing for your own application.

Chapter 9, Razor Components and Data Binding, clarifies what a Razor component is and how to use data binding in a Razor component. You will learn how to create a Razor class library and how to refine existing Razor code to create reusable Razor components.

Chapter 10, Advanced Topics in Creating a Razor Components, brings in some more advanced topics on Razor components. You will learn how to use templated Razor components and built-in Razor components. You will also learn what data validation is and how to perform data validation using built-in components.

Chapter 11, Unit Test Development using xUnit, presents the unit test frameworks available for .NET MAUI. You will learn how to use xUnit and bUnit to develop unit test cases. You will also learn how to create unit test cases for the .NET class and how to create unit test cases for Razor components using bUnit.

Chapter 12, Preparing for Deployment in App Stores, discusses how to prepare packages for app stores and how to set up a CI/CD workflow using GitHub Actions. You will learn how to create packages for Google Play, the App Store, and Microsoft Store. You will also learn how to automate the package creation process using GitHub Actions.

To get the most out of this book

After you have read the first chapter, you can continue with *Part 1* or move on to *Part 2*. In the first part of this book, we discuss classic .NET MAUI app development using a XAML UI. In the second part of this book, we introduce Blazor Hybrid app development, which is new in .NET MAUI. In the third part, we introduce unit tests and deployment.

Both Windows and macOS computers are necessary to build the projects in this book. Visual Studio 2022 and the .NET 6 SDK are used in this book. To build iOS and macOS targets on Windows, you need to connect to a network-accessible Mac, referring to the following Microsoft document:

<https://learn.microsoft.com/en-us/dotnet/maui/ios/pair-to-mac?view=net-maui-6.0>

Software/hardware covered in the book	OS requirements
Visual Studio 2022	Windows
Visual Studio 2022 for Mac	macOS

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development>. If there's an update to the code, it will be updated in the GitHub repository.

My working repository is <https://github.com/shugaoye/PassXYZ.Vault2>.

I will update the source code in my working repository first and then push the commits to Packt repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: <https://packt.link/nvY4N>.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Blazor apps are built using Razor components. The first Razor component in our app is `Main` and it is defined in `Main.razor`.”

A block of code is set as follows:

```
private async Task<bool> UpdateItemAsync(string key, string
value)
{
    if (listGroupItem == null) return false;
    if (string.IsNullOrEmpty(key) || string.IsNullOrEmpty(value))
        return false;

    listGroupItem.Name = key;
    listGroupItem.Notes = value;

    if (_isNewItem) {...}
    else {...}
    StateHasChanged();
}
```

```
    return true;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,VoiceMail(u100)
exten => s,102,VoiceMail(b100)
exten => i,1,VoiceMail(s0)
```

Any command-line input or output is written as follows:

```
git clone -b chapter09
https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-
Application-Development
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “We can right-click on the project node and select **Add -> New Item... -> Razor Component** in the project template.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *.NET MAUI Cross-Platform Application Development*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content..

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?
Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781800569225>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1:

Exploring .NET MAUI

In the first part of this book, we will learn about .NET MAUI programming. We will start with the introduction of .NET MAUI and its ancestor Xamarin.Forms. After that, we will create a code base using the Visual Studio template for our application. We will build a password manager app called PassXYZ.Vault step by step in this book. During the development of this app, we will introduce user interface design using XAML, the MVVM pattern, data binding, the shell, dependency injection, and so on. By the end of *Part 1*, we will have a fully functional password manager application.

This section comprises the following chapters:

- *Chapter 1, Getting Started with .NET MAUI*
- *Chapter 2, Building Our First .NET MAUI App*
- *Chapter 3, User Interface Design with XAML*
- *Chapter 4, Exploring MVVM and Data Binding*
- *Chapter 5, Navigation using .NET MAUI Shell and NavigationPage*
- *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*

Getting Started with .NET MAUI

Since the release of .NET 5, Microsoft has been trying to unify different .NET implementations into one .NET release. .NET Multi-platform App UI (or .NET MAUI) is an effort to provide a unified cross-platform UI framework. We will learn how to use .NET MAUI to develop cross-platform applications in this book.

The following is what we will learn in this chapter:

- An overview of cross-platform technologies
- A comparison of cross-platform technologies (.NET, Java, and JavaScript)
- The .NET landscape and the history of Xamarin
- .NET MAUI features
- .NET MAUI Blazor apps
- A development environment setup

If you're new to .NET development, this chapter will help you to understand the .NET landscape. For Xamarin developers, many topics in this book may sound familiar, and this chapter will give you an overview of what we will discuss in this book.

An overview of cross-platform technologies

Before discussing cross-platform technologies, let's review the application development landscape first to understand the different cross-platform technologies better.

.NET MAUI is a cross-platform development framework from Microsoft for building apps, targeting both mobile and desktop form factors on Android, iOS, macOS, Windows, and Tizen.

Generally, software development can be divided into two categories – systems programming and application programming. Application programming aims to produce software that provides services to the user directly, whereas system programming aims to produce software and software platforms that provide services to other software. In the .NET domain, the development of the .NET platform itself belongs to systems programming, whereas the application development on top of the .NET platform belongs to application programming.

The design or architecture in a modern system includes the client and server side of software, which we can refer to as the frontend and backend.

For the software on the client side, we can further divide it into two categories – native applications and web applications.

Native applications

In native application development, we usually refer to application development for a particular operating system. With desktop applications, this could be Windows applications, macOS applications, or Linux applications. With mobile applications, this could be Android or iOS.

When we develop a native application, we have to develop it for each platform (Windows, Linux, Android, or macOS/iOS). We need to use different programming languages, tools, and libraries to develop each of them individually.

Web applications

Web application development has gone through several generations of evolution over the past few decades, from a Netscape browser with static web pages to a modern **single-page application (SPA)** using JavaScript frameworks (such as React or Angular). In web application development, JavaScript and various JavaScript-based frameworks dominate the market. In the .NET ecosystem, Blazor is trying to catch up in this area.

Backend services

Both native applications and web applications usually need some backend services to access business logic or a database. For backend development, many languages and frameworks can be used, such as Java/Spring, .NET, Node.js, Ruby on Rails, or Python/Django. Usually, native applications and web applications can share the same backend service. Java and .NET are the most popular choices for backend service developments.

Cross-platform technologies

Technologies used in web application development and backend services development are not platform-specific and can be used on different platforms as they are. When we talk about cross-platform development, we usually refer to native application development. In native application development,

cross-platform development technologies can help to reduce costs and improve efficiency. The most popular cross-platform development technologies in this category include Flutter, .NET MAUI/Xamarin, and React Native. *Table 1.1* provides an overview of available cross-platform technologies and alternative solutions from Microsoft. The technologies listed here are not exhaustive. I just want to give you a feeling of what kind of technologies exist in each category and what Microsoft solution can be used as an alternative.

Category	Cross-platform technologies		Microsoft solution
	Language	Framework	
Web application	JavaScript	React, Angular, or Vue	Blazor/Razor Pages
Native application	JavaScript	React Native, Cordova, Ionic, Electron, or NW.js	.NET MAUI/Blazor/Xamarin
	Dart	Flutter	
	Java/Kotlin	Swing/Codename One	
Backend services	Java	Spring	ASP.NET Core
	JavaScript	Node.js	
	Python	Diango/Flask/Tornado	

Table 1.1: A comparison of languages and frameworks with Microsoft solutions

There is no best choice of cross-platform tool or framework. The final choice is usually decided according to business requirements. However, from the preceding table, we can see that the .NET ecosystem provides a full spectrum of tools for your requirements. The development team for a large system usually requires people with experience in different programming languages and frameworks. With .NET, the complexity of programming languages and frameworks can be dramatically simplified.

A comparison of .NET, Java, and JavaScript

We had an overview of the tools and frameworks used in web apps, native apps, and backend services development. If we look at a higher level, that is, at the .NET ecosystem level, the ecosystem of Java or JavaScript can match almost what we have in a .NET solution. Java, JavaScript, or .NET solutions can provide tools or frameworks at nearly all layers. It would be interesting to compare Java, JavaScript, and .NET at a higher level.

Java is developed as a language with the goal to *write once and run anywhere*. It is built around the Java programming language and the **Java Virtual Machine (JVM)**. The JVM is a mechanism to run on supported platforms that helps to remove platform dependency for developers. With this cross-platform capability, Java becomes a common choice for cross-platform applications and services development.

JavaScript is a language created for web browsers, and its capability is extensive due to the demands of web development. The limitation of JavaScript is that it is a scripting language, so it lacks the language features that can be found in Java or C#. However, this limitation doesn't limit its usage and popularity. *Table 1.2* offers a comparison of three technologies:

Area of comparison	.NET	Java	JavaScript
Programming languages	C#, F#, VB, C++, PHP, Ruby, Python, and more	Java, Kotlin, Clojure, Groovy, Scala, and more	JavaScript, TypeScript, CoffeeScript, and more
Runtime	CLR	JVM	V8/SpiderMonkey/JavaScriptCore
Supported IDE	Microsoft Visual Studio, Rider, MonoDevelop, and Visual Studio Code	Eclipse, IntelliJ Idea, Oracle NetBeans, and Oracle JDeveloper	Visual Studio Code, Webstorm, and Atom
Frontend framework	ASP.NET Core Razor/Blazor	Only supports server-side rendering such as JSP or Thymeleaf	React, Angular, or Vue
Desktop apps	WinForms, Win UI, WPF, UWP, and more	Swing, JavaFX, and more	Electron, NW.js, and more
Mobile apps	.NET MAUI/Xamarin	Codename One	React Native, Cordova, Ionic, and more
Backend framework	ASP.NET Core	The Spring Framework	Node.js

Table 1.2: A comparison of Java, JavaScript, and .NET

From *Table 1.2*, we can see that both .NET and Java have a good infrastructure to support multiple languages. JavaScript has its limitation as a scripting language, so TypeScript and CoffeeScript were invented to enhance it. TypeScript was developed by Microsoft to bring modern object-oriented language features to JavaScript. TypeScript is compiled into JavaScript for execution, so it can work well with existing JavaScript libraries.

Java is built around the JVM while .NET is built around the **Common Language Runtime (CLR)** and the **Common Type System (CTS)**. With the CTS and CLR as the core of a .NET implementation, it supports multiple languages naturally with the capability to share a **Base Class Library (BCL)** in all supported languages.

While there are multiple languages that use the JVM as the abstraction layer for cross-platform capability, the interoperability between Java-derived languages is not at the same level as .NET languages. All .NET languages are built on one architecture and share the same BCL, while Java languages, such as Java, Kotlin, or Scala, are developed separately for very different purposes.

This comparison helps us to choose or evaluate a tech stack for cross-platform development. As a .NET MAUI developer, this analysis can help you understand your choice better. To understand where .NET MAUI is located in the .NET ecosystem, let's have a quick overview of the history of the .NET landscape in the next section.

Exploring the .NET landscape

Before we dive into the details of .NET MAUI, let's have an overview of the .NET landscape. This section is relevant if you are new to .NET. If you are a .NET developer, you can skip this section.

Since Microsoft introduced the .NET platform, it has evolved from a proprietary software framework for Windows to a cross-platform and open source platform.

There are many ways to look at the .NET technology stack. Basically, it contains the following components:

- Common infrastructure (Compiler and tools suite)
- BCLs
- Runtime (**Windows Runtime (WinRT)** or Mono)

.NET Framework

The history of .NET history begins with .NET Framework. It is a proprietary software framework developed by Microsoft that runs primarily on Microsoft Windows. .NET Framework started as a future-oriented application framework to standardize the software stack in the Windows ecosystem. It is built around a **Common Language Infrastructure (CLI)** and C#. Even though the primary programming language is C#, it is designed to be a language-agnostic framework. Supported languages can share the same CTS and CLR. Most Windows desktop applications are developed using .NET Framework, and it is shipped as a part of the Windows operating system.

Mono

The first attempt to make .NET an open source framework was made by a company called Ximian. When the CLI and C# were ratified by ECMA in 2001 and ISO in 2003, it provided a potential opportunity for independent implementations.

In 2001, the open source project Mono was launched, aimed at implementing .NET Framework on Linux desktop software.

Since .NET Framework was a proprietary technology at that time, .NET Framework and Mono had their own compiler, BCL, and runtime.

Over time, Microsoft moved toward open source, and .NET source code became open source. The Mono project adopted some source code and tools from the .NET code base.

At the same time, Mono projects went through many changes as well. At the time that Mono was owned by the Xamarin company, Xamarin developed the Xamarin platform based on Mono to support the .NET platform on Android, iOS, **Universal Windows Platform (UWP)**, and macOS. In 2016, Microsoft acquired Xamarin, which became the cross-platform solution in the .NET ecosystem.

.NET Core

Before the acquisition of Xamarin, Microsoft has already started work to make .NET a cross-platform framework. The first attempt was the release of .NET Core 1.0 in 2016. .NET Core is a free and open source framework, available for Windows, Linux, and macOS. It can be used to create modern web apps, microservices, libraries, and console applications. Since .NET Core applications can run on Linux, we can build microservices using containers and cloud infrastructure.

After .NET Core 3.x was released, Microsoft worked toward integrating and unifying .NET technology on various platforms. This unified version was to supersede both .NET Core and .NET Framework. To avoid confusion with .NET Framework 4.x, this unified framework was named .NET 5. Since .NET 5, a common BCL can be used on all platforms. In .NET 5, there are still two runtimes, which are **WinRT** (used for Windows) and the Mono runtime (used for mobile and macOS).

In this book, we use will the .NET 6 release.

.NET Standard and portable class libraries

Before the .NET 5 releases, with .NET Framework, Mono, and .NET Core, we had a different subset of BCLs on different platforms. In order to share code between different runtimes or platforms, a technique called **Portable Class Libraries (PCLs)** was used. When you create a PCL, you have to choose a combination of platforms that you want to support. The level of compatibility choices is decided by the developers. If you want to reuse any PCL, you must carefully study the list of platforms that can be supported.

Even though a PCL provides a way to share code, it cannot resolve compatibility issues nicely. To overcome the compatibility issues, Microsoft introduced .NET Standard.

.NET Standard is not a separate .NET release but instead a specification of a set of .NET APIs that must be supported by most .NET implementations (.NET Framework, Mono, .NET Core, .NET 5 or 6, and so on).

After .NET 5 and later versions, a unified BCL is available, but .NET Standard will be still part of this unified BCL. If your applications only need to support .NET 5 or later, you don't really need to care too much about .NET Standard. However, if you want to be compatible with old .NET releases, .NET Standard is still the best choice for you. We will use .NET Standard 2.0 in this book to build our data model, since this is a version that can support most existing .NET implementations and all future .NET releases.

There will be no new versions of .NET Standard from Microsoft, but .NET 5, .NET 6, and all future versions will continue to support .NET Standard 2.1 and earlier. *Table 1.3* shows the platforms and versions that .NET Standard 2.0 can support, and this is also the compatible list for our data model in this book.

.NET implementation	Version support
.NET and .NET Core	2.0, 2.1, 2.2, 3.0, 3.1, 5.0, and 6.0
.NET Framework 1	4.6.1 2, 4.6.2, 4.7, 4.7.1, 4.7.2, and 4.8
Mono	5.4 and 6.4
Xamarin.iOS	10.14 and 12.16
Xamarin.Mac	3.8 and 5.16
Xamarin.Android	8.0 and 10.0
UWP	10.0.16299, TBD
Unity	2018.1

Table 1.3: .NET Standard 2.0-compatible implementations

The open-source project `KPCLib` is a .NET Standard 2.0 library, and we will use it in our app. In *Table 1.3*, we can see that .NET Standard libraries can be used in both Xamarin and .NET MAUI apps.

Using Xamarin for mobile development

As we mentioned in an earlier section, Xamarin was part of the Mono project and was an effort to support .NET on Android, iOS, and macOS. Xamarin.Forms is a cross-platform UI framework from Xamarin. .NET MAUI is an evolution of Xamarin.Forms. Before we discuss .NET MAUI and Xamarin.Forms, let us review the following diagram of Xamarin implementation on various platforms.

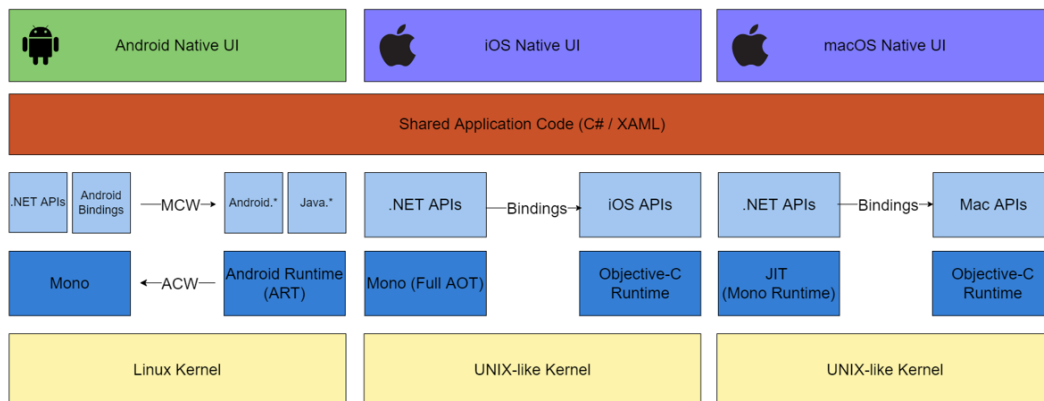


Figure 1.1: Xamarin implementations

Figure 1.1 shows the overall architecture of Xamarin. Xamarin allows developers to create native UIs on each platform and write business logic in C# that can be shared across platforms.

On supported platforms, Xamarin contains bindings for nearly the entire underlying platform SDKs. Xamarin also provides facilities for directly invoking the Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of third-party code. You can use existing Android, iOS, or macOS libraries written in Objective-C, Swift, Java, or C/C++.

The Mono runtime is used as the .NET runtime on these platforms. It has two modes of operation – **Just-in-Time (JIT)** and **Ahead-of-Time (AOT)**. JIT compilation generates code dynamically as it is executed. In AOT compilation mode, Mono precompiles everything, so it can be used on operating systems where dynamic code generation is not possible.

As we can see in *Figure 1.1*, JIT can be used on Android and macOS, while AOT is used for iOS where dynamic code generation is not allowed.

There are two ways to develop native applications using Xamarin.

You can develop native applications just like Android, iOS, or macOS developers, using native APIs on each platform. The difference is that you use .NET libraries and C# instead of the platform-specific language and libraries directly. The advantage of this approach is you can use one language and share a lot of components through the .NET BCL, even if you work on different platforms. You can also leverage the power of underlying platforms like native application developers.

If you want to reuse code on the user interface layer, you can use Xamarin.Forms instead of the native UI.

Xamarin.Forms

Xamarin.Android, Xamarin.iOS, and Xamarin.Mac provide a .NET environment that exposes almost the entire original SDK capability on their respective platforms. For example, as a developer, you have almost the same capability with Xamarin.Android as you do with the original Android SDK. To further improve code sharing, an open source UI framework, Xamarin.Forms, was created. Xamarin.Forms includes a collection of cross-platform user interface components. The user interface design can be implemented using the XAML markup language, which is similar to Windows user interface design in WinUI or WPF.

Xamarin.Essentials

Since Xamarin exposes the capability of the underlying platform SDKs, you can access device features using the .NET API. However, the implementation is platform-specific. For example, when you use a location service on Android or iOS, the .NET API can be different. To further improve code sharing across platforms, Xamarin.Essentials can be used to access native device features. Xamarin.Essentials provides a unified .NET interface for native device features. If you use Xamarin.Essentials instead of native APIs, your code can be reused across platforms.

Some examples of functionalities provided by Xamarin.Essentials include the following:

- Device info
- The filesystem
- An accelerometer
- A phone dialer
- Text-to-speech
- Screen lock

Using Xamarin.Forms together with Xamarin.Essentials, most implementations, including business logic, user interface design, and some level of device-specific features, can be shared across platforms.

Comparing user interface design on different platforms

Most modern application development on various platforms uses the **Model-View-Controller (MVC)** design pattern. To separate the business logic and user interface design, there are different approaches used on Android, iOS/macOS, and Windows. On all the platforms involved, even though the programming languages used are different, they all use XML-based markup language to design user interfaces.

On an iOS/macOS platform, developers can use Interface Builder in XCode to generate `.storyboard` or `.xib` files. Both are XML-based script files used to keep user interface information, and this script is interpreted at runtime together with Swift or Objective-C code to create the user interface. In 2019, Apple announced a new framework, SwiftUI. Using SwiftUI, developers can build user interfaces using the Swift language in a declarative way directly.

On the Android platform, developers can use **Layout Editor** in Android Studio to create a user interface graphically and store the result in layout files. The layout files are in the XML format as well and can be loaded at runtime to create the user interface.

On the Windows platform, **Extensible Application Markup Language (XAML)** is used in user interface design. **XAML** is an XML-based language used for user interface design on the Windows platform. For a WPF or UWP application, the XAML Designer can be used for user interface design. In .NET MAUI, the XAML-based UI is the default application UI. Another pattern, the **Model View Update (MVU)** pattern, can also be used. In the MVU pattern, the user interface is implemented in C# directly without XAML. The coding style of MVU is similar to SwiftUI.

Even though SwiftUI on Apple platforms or MVU in .NET MAUI can be used, but the classic user interface implementation is the XML-based markup language. Let us do a comparison in *Table 1.4*.

Platform	IDE	Editor	Language	File extension
Windows	Visual Studio	XAML Designer	XAML/C#	.xaml

Platform	IDE	Editor	Language	File extension
Android	Android Studio	Layout Editor	XML/Java/Kotlin	.layout
iOS/macOS	Xcode	Interface Builder	XML/Swift/Objective C	.storyboard or .xib
.NET MAUI/ Xamarin.Forms	Visual Studio	N.A.	XAML/C#	.xaml
.NET MAUI Blazor			Razor/C#	.razor

Table 1.4: A comparison of user interface design

In *Table 1.4*, we can see a comparison of user interface design on different platforms.

.NET MAUI and Xamarin.Forms use a dialect of XAML to design user interfaces on all supported platforms. For .NET MAUI, we have another choice for user interface design, which is Blazor. We will discuss Blazor later in this chapter.

In Xamarin.Forms, we create user interfaces in XAML and code-behind in C#. The underlying implementation is still the native controls on each platform, so the look and feel of Xamarin.Forms applications are the same as native ones.

Some examples of features provided by Xamarin.Forms include the following:

- XAML user interface language
- Data binding
- Gestures
- Effects
- Styling

Even though we can share almost all UI code with Xamarin.Forms, we still need to handle most of the resources used by an application in each platform individually. These resources could be images, fonts, or strings. In the project structure of Xamarin.Forms, we have a common .NET standard project and multiple platform-specific projects. Most of the development work will be done in the common project, but the resources are still handled in the platform-specific projects separately.

Moving to .NET MAUI

With the .NET unification, Xamarin has become a part of the .NET platform, and Xamarin.Forms integrates with .NET in the form of .NET MAUI.

.NET MAUI is a first-class .NET citizen with the `Microsoft.Maui` namespace.

Making the move to .NET MAUI is also an opportunity for Microsoft to redesign and rebuild Xamarin.Forms from the ground up and tackle some of the issues that have been lingering at a lower level. Compared to Xamarin.Forms, .NET MAUI uses a single project structure, supports hot reloads better, and supports MVU and Blazor development patterns.

From *Figure 1.2*, we can see that there is a common BCL for all supported operating systems. Under the BCL, there are two runtimes, WinRT and the Mono Runtime, according to the platform. For each platform, there is a dedicated .NET implementation to provide full support for native application development.

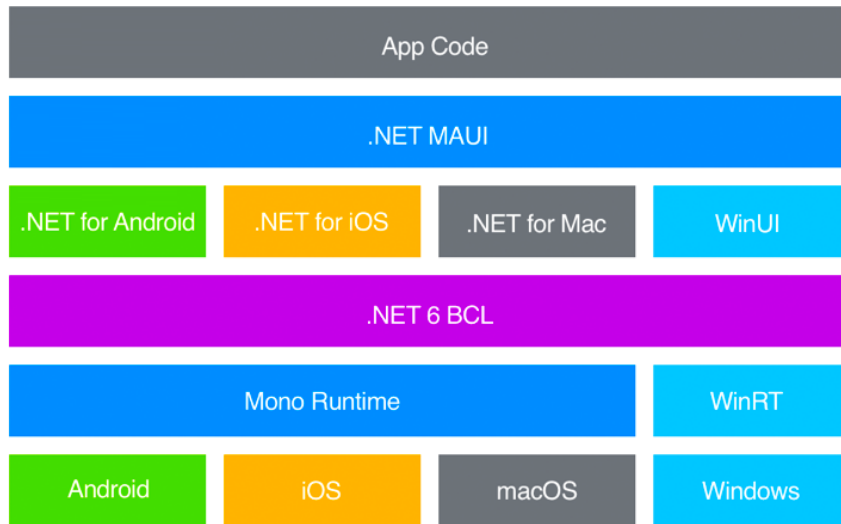


Figure 1.2: .NET MAUI architecture

Comparing to Xamarin.Forms, we can see from *Table 1.5*, there are many improvements in .NET MAUI.

.NET MAUI uses a single project structure to simplify project management. We can manage resources, dependency injection, and configurations in one location instead of managing them separately per platform.

.NET MAUI is fully integrated as part of .NET, so we can create and build projects using the .NET SDK command line. In this case, we have more choices in terms of development environments.

	.NET MAUI	Xamarin.Forms
Project structure	Single project	Multiple projects
Resource management	One location for all platforms	Managed per platform

	.NET MAUI	Xamarin.Forms
Fully integrated with .NET	Namespace in <code>Microsoft.Maui</code> and other IDEs can be chosen beside Visual Studio. Command-line support. We can create, build, and run in a console: <pre>dotnet new maui dotnet build -t:Run -f net6.0-android dotnet build -t:Run -f net6.0-ios dotnet build -t:Run -f net6.0-maccatalyst</pre>	Namespace in <code>Xamarin.Forms</code> and it uses Visual Studio as an IDE
Design improvement	• Configuration through .NET Generic Host • Dependency injection support	• Configuration scattered in different locations
MVU pattern	A modern type of UI implementation	No
Blazor Hybrid	Support through <code>BlazorWebView</code>	No

Table 1.5: .NET MAUI improvement

In *Table 1.5*, we can see that .NET MAUI supports application configuration using .NET generic host, can work with multiple IDE environments, supports dependency injection, and can use the MVVM toolkit, etc. It also supports the MVU pattern and Blazor Hybrid UI. Next, we will look at the Blazor Hybrid app.

.NET MAUI Blazor apps

In *Table 1.4*, where we compared the user interface design options on different platforms, we mentioned that there is another way to design cross-platform user interfaces in .NET MAUI, which is Blazor.

Released in ASP.NET Core 3.0, Blazor is a framework for building an interactive client-side web UI with .NET. With .NET MAUI and Blazor, we can build cross-platform apps in the form of Blazor Hybrid apps. This way, the boundary between a native application and a web application becomes blurred. .NET MAUI Blazor Hybrid apps enable Blazor components to be integrated with native platform features and UI controls. The Blazor components have full access to the native capabilities of a device.

The way to use the Blazor web framework in .NET MAUI is through a `BlazorWebView` component. .NET MAUI Blazor enables both native and web UIs in a single application, and they can co-exist in a single view. With .NET MAUI Blazor, applications can leverage the Blazor component model (Razor components), which uses HTML, CSS, and the Razor syntax. The Blazor part of an app can reuse components, layouts, and styles that are used in an existing regular web app. `BlazorWebView` can be composed alongside native elements; additionally, they leverage platform features and share states with their native counterparts.

Choosing XAML versus Razor in .NET MAUI

To design the user interface of your .NET MAUI application, you have a few choices for implementation:

- **XAML:** Implement user interface in XAML that is only similar to Xamarin.Forms. We can also choose the MVU pattern to use C# code to create and style UI elements directly. No matter whether you choose XAML or C# code directly, the underlying implementation is the same.
- **Blazor:** Implement a user interface in Razor Pages, which is similar to web application development.
- **Blazor Hybrid app:** Use both XAML and Razor Pages in your application.

It's your decision how you want to design your application. You can choose one of the preceding options or mix XAML and the Blazor UI according to the best fit. To develop Blazor Hybrid apps, you should be able to use most of the existing Blazor libraries directly. Blazor provides good JavaScript interoperability, and you can use your favorite JavaScript library in your development.

Development environment setup

Both Windows and macOS can be used for .NET MAUI development, but you won't be able to build all targets with only one of them. You will need both Windows and Mac computers to build all targets. In this book, the Windows environment is used to build and test Android and Windows targets. iOS and macOS targets are built on a Mac computer.

.NET MAUI app can target the following platforms:

- Android 5.0 (API 21) or higher
- iOS 10 or higher
- macOS 10.13 or higher, using Mac Catalyst
- Windows 11 and Windows 10 version 1809 or higher, using **Windows UI Library (WinUI)** 3

.NET MAUI Blazor apps use the platform-specific WebView control, so they have the following additional requirements:

- Android 7.0 (API 24) or higher
- iOS 14 or higher
- macOS 11 or higher, using Mac Catalyst

.NET MAUI build targets of Android, iOS, macOS, and Windows can be built using Visual Studio on a Windows computer. In this environment, a networked Mac is required to build iOS and macOS targets. Xcode must be installed on the paired Mac to debug and test an iOS MAUI app in a Windows development environment.

.NET MAUI targets of Android, iOS, and macOS can be built and tested on macOS.

Target platform	Windows	macOS
Windows	Yes	No
Android	Yes	Yes
iOS	Yes (pair to Mac)	Yes
macOS	Build only (pair to Mac)	Yes

Table 1.6: The development environment of .NET MAUI

Please refer to Table 1.6 to find out the build configurations on Windows and macOS.

Installing .NET MAUI on Windows

.NET MAUI can be installed as part of Visual Studio 2022. The Visual Studio Community edition is free, and we can download it from the Microsoft website at <https://visualstudio.microsoft.com/vs/community/>.

After launching Visual Studio Installer, we will see a screen similar to the one shown in *Figure 1.3*. Please select **.NET Multi-platform App UI development** and **.NET desktop development** in the list of options. We also need to select **ASP.NET and web development** for the .NET MAUI Blazor app, which will be covered in *Part 2* of this book.

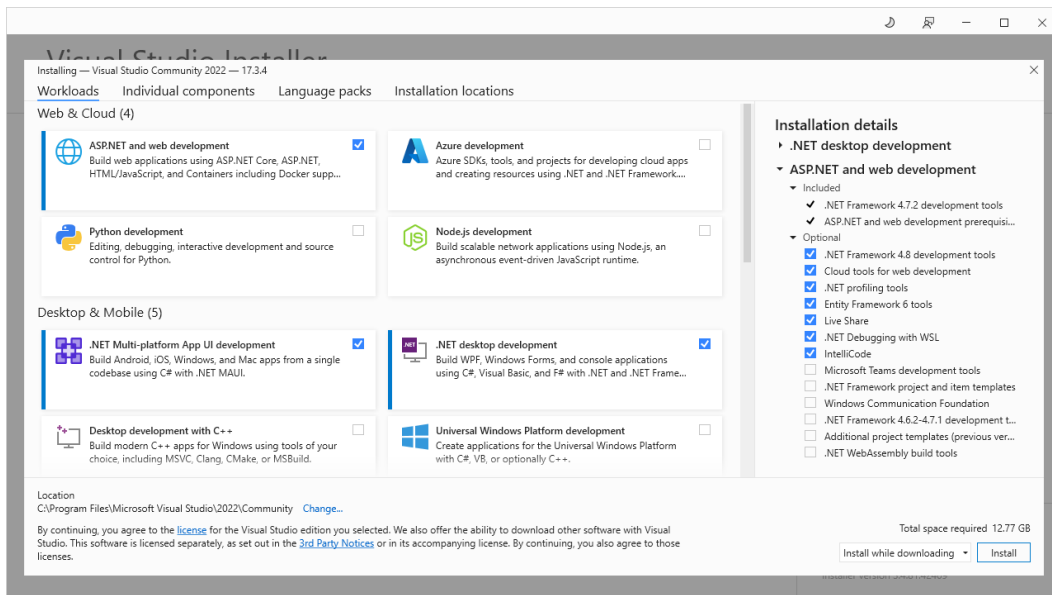


Figure 1.3: Visual Studio 2022 installation

After the installation is completed, we can check the installation from the command line using the following dotnet command:

```
C:\>dotnet workload list
Installed Workload Ids  Manifest Version  Installation Source
-----
-----
maui-windows           6.0.486/6.0.400   VS 17.3.32811.315
maui-maccatalyst       6.0.486/6.0.400   VS 17.3.32811.315
maccatalyst            15.4.446-ci.-release-6-0-
4xx.446/6.0.400        VS 17.3.32811.315
maui-ios               6.0.486/6.0.400   VS 17.3.32811.315
ios                    15.4.446-ci.-release-6-0-
4xx.446/6.0.400        VS 17.3.32811.315
maui-android           6.0.486/6.0.400   VS 17.3.32811.315
android                32.0.448/6.0.400   VS 17.3.32811.315
Use `dotnet workload search` to find additional workloads to
install.
```

We are ready to create, build, and run a .NET MAUI app on Windows.

Installing .NET MAUI on macOS

The installation of the Visual Studio Community edition is similar to what we have done on Windows. The installation package can be downloaded from the same link.


```
maccatalyst      15.4.453/6.0.400      SDK 6.0.400
maui              6.0.536/6.0.400      SDK 6.0.400
tvos              15.4.453/6.0.400      SDK 6.0.400
android           32.0.465/6.0.400      SDK 6.0.400
Use `dotnet workload search` to find additional workloads to
install.
```

We are ready to create, build, and run a .NET MAUI app on macOS.

What you will learn in this book

In this book, you will learn how to develop cross-platform applications using .NET MAUI. The following are the topics covered in this book:

- The .NET ecosystem and .NET MAUI
- User interface design using XAML
- Data binding using the MVVM design pattern
- .NET MAUI Shell navigation and design navigations using routes
- Dependency injection
- Exploring Blazor to design a user interface using the .NET MAUI Blazor Hybrid app
- Using `xUnit.net` to write unit test cases for .NET classes and `bUnit` to implement test cases for Razor Pages
- Publishing an application to Google Play, Apple Store, and Microsoft Store

The app that we will build in this book

Throughout the book, we will learn about .NET MAUI programming by building a password manager app. KeePass is an open source password manager used by many people. However, it is an application built for .NET Framework, so it can run on Windows only. KeePass has a library, `KeePassLib`, which implements most of the logic for encrypted database management. This is a popular database format for password management. For example, IntelliJ IDEA uses the KeePass database to store passwords used during development phases.

To make `KeePassLib` a cross-platform library, I ported `KeePassLib` to .NET Standard 2.0 as the open source `KPCLib` project. The source code can be found at <https://github.com/passxyz/KPCLib>.

I published an app, `PassXYZ.Vault`, developed using `Xamarin.Forms` and `KPCLib` in app stores. In this book, we will rewrite `PassXYZ.Vault` together, using .NET MAUI. You can learn about cross-platform programming through the progressive development of this app until it is published in app stores.

Summary

In this chapter, we started with an overview of cross-platform technologies. We compared a .NET solution with other cross-platform technologies. After that, we went through the .NET landscape. I explained the relationship between .NET Framework, Mono, and .NET Core. Then, we discussed Xamarin and .NET MAUI. We reviewed the difference between Xamarin.Forms and .NET MAUI. One important feature in .NET MAUI is that we can use Blazor together with XAML user interface design. Then, we had an overview of .NET MAUI Blazor. Finally, we set up a development environment for the rest of the chapters.

In the next chapter, we will explore how to build a .NET MAUI application from scratch.

Further reading

- **.NET MAUI:** You can find more information about .NET MAUI in Microsoft's official documentation: <https://docs.microsoft.com/en-us/dotnet/maui/>
- **KeePass:** The official website for KeePass can be found at <https://keepass.info/>

2

Building Our First .NET MAUI App

In this chapter, we will create a new .NET MAUI project and make the necessary changes so that we can use it in the subsequent development. The app that we will develop is a password manager app. We will add features to it gradually in the coming chapters. When we complete *Part 1*, we will have a functional password manager app. In this chapter, we will create the app using the Visual Studio template, and initialize the resources of the application. After that, we will build and test it on supported platforms. To use Shell in our app, we will create a new Xamarin.Forms project with Shell support and then migrate it to our .NET MAUI project.

The following topics will be covered in this chapter:

- Setting up a new .NET MAUI project
- App startup and lifecycle management
- Configuring resources
- Creating a new Xamarin.Forms project with Shell
- Migrating this Xamarin.Forms project to .NET MAUI

Technical requirements

To test and debug the source code in this chapter, you need Visual Studio 2022 installed on both Windows and macOS. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

Managing the source code in this book

Since we develop a password manager app incrementally in this book, the source code of each chapter is built on top of the previous chapters. To keep working on continuous improvement, we will have

separate branches to keep the source code of each chapter. If you want to clone the source code of all chapters in one command, you can clone from the main branch. In the main branch, we have all chapters in separate folders. If you don't want to use Git, you can also download the source code as a compressed file from the release area, as shown in the following diagram (Figure 2.1):

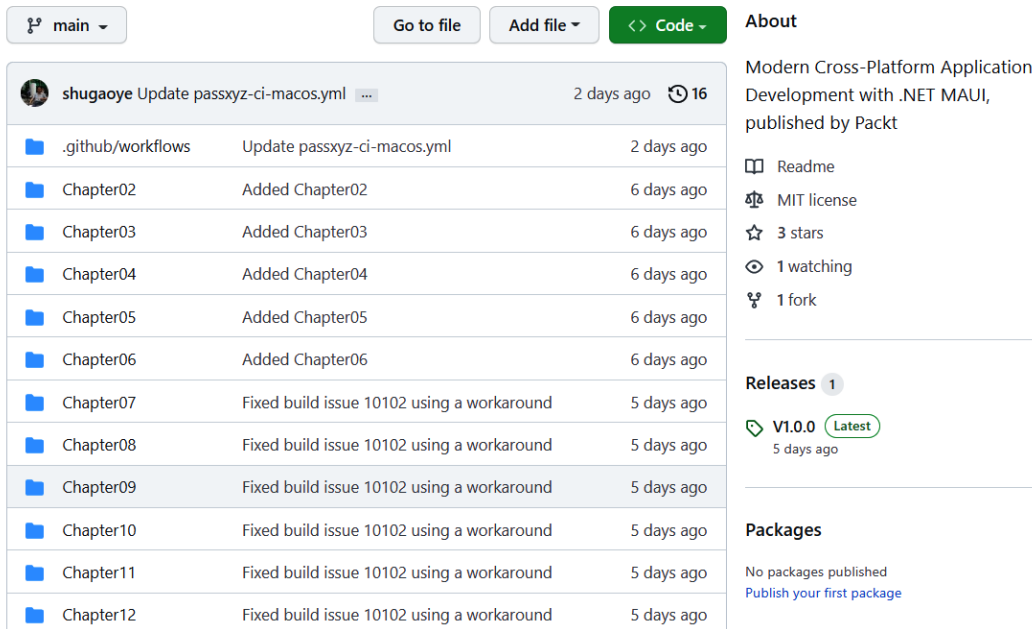


Figure 2.1: Source code in GitHub

Since new .NET MAUI releases may be available from time to time, the Git tags and versions in the release area will be updated according to the new .NET MAUI releases and bug fixes.

The source code of this book can be found in the following GitHub repository:

<https://github.com/PacktPublishing/Modern-Cross-Platform-Application-Development-with-.NET-MAUI>

There are three ways to download the source code in this book:

- Download the source code in a compressed file.

The source code can be downloaded in the release area, or use the following URL:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/releases/tag/V1.0.0>

The release tag may be changed when a new release is available.

- Clone the source code of one chapter.

To check out the source code of a chapter, you can use the following command as an example:

```
$ git clone https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development.git -b chapter02
```

- Clone the source code from the main branch.

To check out the source code of all chapters from the main branch, you can use the following command:

```
$ git clone https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development.git
```

Setting up a new .NET MAUI project

We can create a new .NET MAUI project using Visual Studio or the command line.

Creating a new project using Visual Studio

To create a new .NET MAUI project, follow these steps:

1. Launch Visual Studio 2022 and select **Create a new project** on the startup screen. This will open the **Create a new project** wizard.

In the top-middle section of the screen, there is a search box. We can type `Maui` in the search box, and .NET MAUI-related project templates will be shown (see *Figure 2.2*):

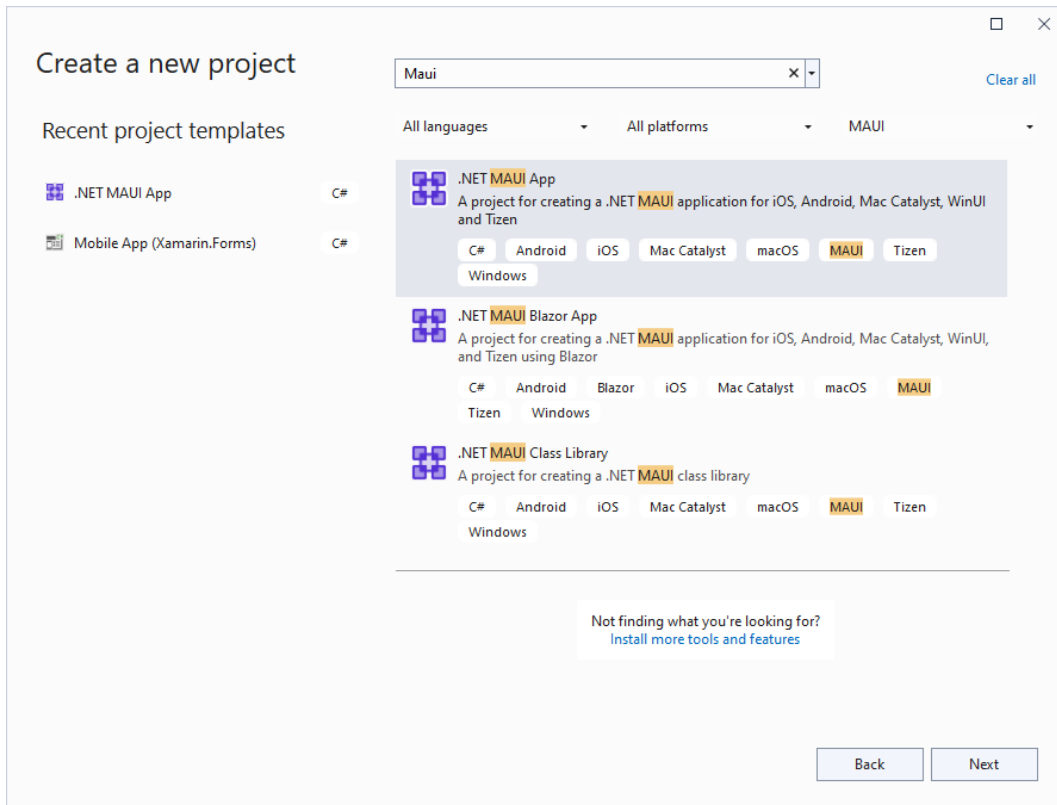
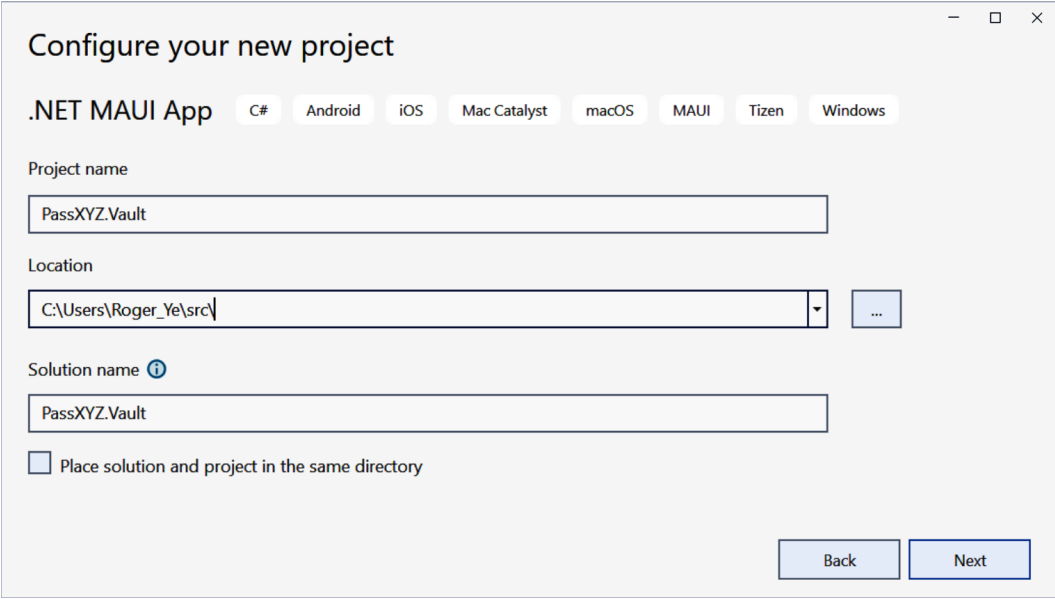


Figure 2.2: New project setup – Create a new project

There are three templates for the .NET MAUI app or library:

- **.NET MAUI App** – This is for a XAML-based .NET MAUI app.
 - **.NET MAUI Blazor App** – This template can be used to create a .NET MAUI Blazor app.
 - **.NET MAUI Class Library** – This is the option to build a .NET MAUI class library. We can build shared components as a .NET MAUI class library when we develop a .NET MAUI app.
2. Let's select **.NET MAUI App** and click the **Next** button; it goes to the next step to configure your new project, as shown in *Figure 2.3*:



Configure your new project

.NET MAUI App C# Android iOS Mac Catalyst macOS MAUI Tizen Windows

Project name

PassXYZ.Vault

Location

C:\Users\Roger_Ye\src\

Solution name ⓘ

PassXYZ.Vault

☐ Place solution and project in the same directory

Back Next

Figure 2.3: New project setup – Configure your new project

3. Enter the project name and solution name as `PassXYZ.Vault` and click the **Next** button. After the project is created, the project structure will look like *Figure 2.4*, displaying the following:
 - *Common files* – In a new project, there are three files included in the template – `App.xaml`, `MainPage.xaml`, and `MauiProgram.cs`. This is the group of files that we will work on throughout the book. They are platform agnostic. Both business logic and UI can be developed here and shared on all platforms.
 - *Platform-specific files* – There are five subfolders (`Android`, `iOS`, `MacCatalyst`, `Windows`, and `Tizen`) in the `Platforms` folder. Since we won't support `Tizen`, we can remove it from our project.

- *Resources* – A variety of resources ranging from images, fonts, splash screens, styles, and raw assets are in the `Resources` folder. These resources can be used in all supported platforms.

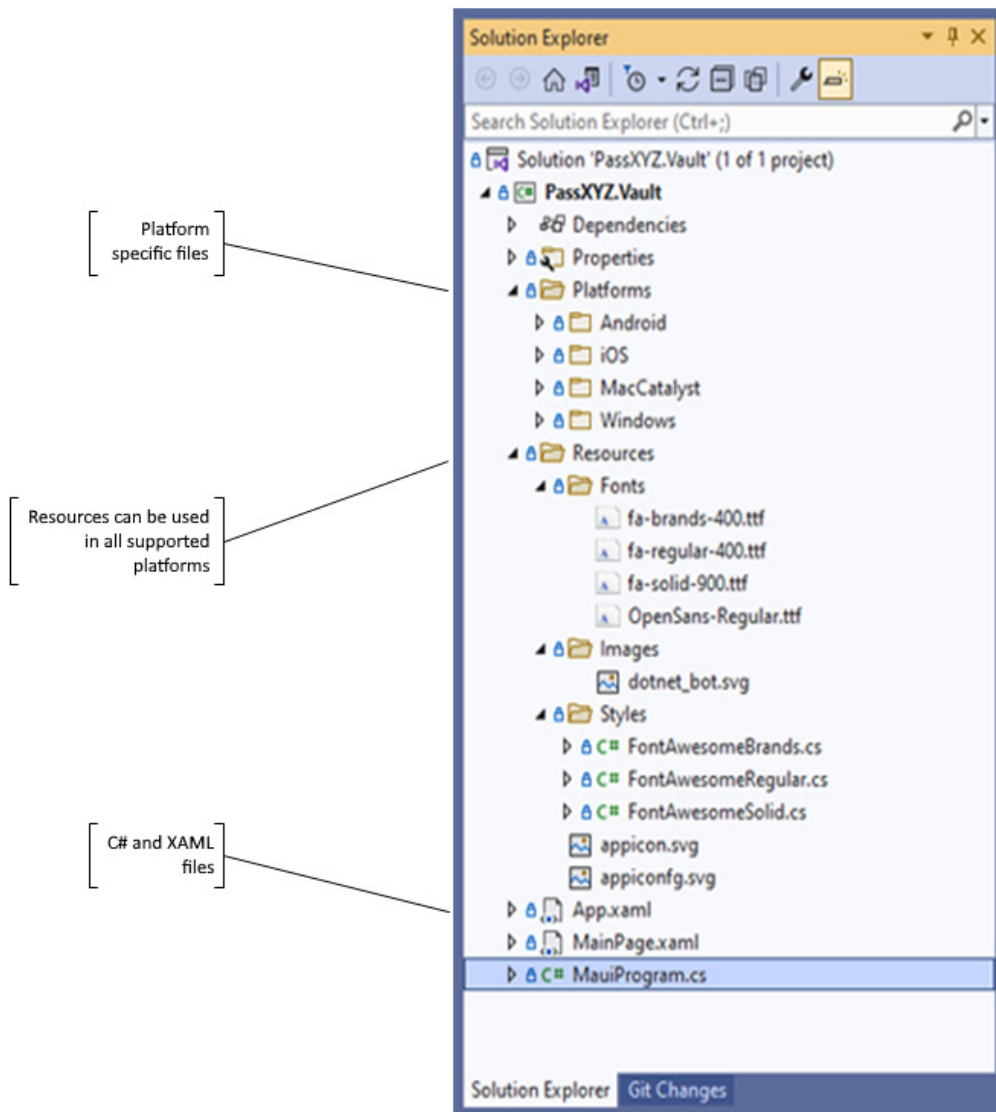


Figure 2.4: .NET MAUI project structure

In the .NET MAUI project, there is only one project structure. Later, we will see that the development of Xamarin.Forms involves multiple projects.

Creating a new project using the dotnet command

Even though we installed .NET MAUI as part of the Visual Studio installation, .NET MAUI can be installed separately using the command line as well. In this way, we can choose other development tools than Visual Studio. We can create and build a .NET MAUI app using the `dotnet` command line. To find out about the installed project templates, we can use the following command:

```
C:\ > dotnet new --list
```

To create a new project using the command line, we can execute the following command:

```
C:\ > dotnet new maui -n "PassXYZ.Vault"
```

The new .NET MAUI project has been created, and we can build and test it now. Before we move to that, let's spend some time having a look at the .NET MAUI app startup code and lifecycle.

App startup and lifecycle

In the .NET MAUI project, the app startup and lifecycle management are handled in the following two files:

- `MauiProgram.cs`
- `App.xaml/App.xaml.cs`

For the app startup and configuration, .NET Generic Host is used. When the application starts, a .NET Generic Host object is created to encapsulate an app's resources and lifetime functionality, such as the following:

- **Dependency injection (DI)**
- Logging
- Configuration
- App shutdown

This enables apps to be initialized in a single location and provides the ability to configure fonts, services, and third-party libraries.

.NET Generic Host

If you are a Xamarin developer, you may not be familiar with .NET Generic Host. In ASP.NET Core, .NET Generic Host is used to encapsulate the resources in an app. In .NET MAUI, the same pattern is borrowed and used for startup and configuration management.

Let's examine the app startup code in *Listing 2.1* (`MauiProgram.cs`):

Listing 2.1: MauiProgram.cs (<https://epa.ms/MauiProgram2-1>)

```
namespace PassXYZ.Vault;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp() ❶
    {
        var builder = MauiApp.CreateBuilder(); ❷
        builder
            .UseMauiApp<App>() ❸
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf",
                    "OpenSansRegular");
            });

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

We can see the following in *Listing 2.1*:

- ❶ In each platform, the entry point is in platform-specific code. The entry point calls the `CreateMauiApp` function, which is a method of the `MauiProgram` static class.
- ❷ Inside `CreateMauiApp`, it calls the `CreateBuilder` function, which is a method of the `MauiApp` static class, and returns a `MauiAppBuilder` instance, which provides a .NET Generic Host interface.
- ❸ The return value of `CreateMauiApp` is a `MauiApp` instance, which is the entry point of your app.

④ The `App` class referenced in the `UseMauiApp` method is the root object of our application. Let's review the definition of the `App` class in *Listing 2.2*:

Listing 2.2: App.xaml.cs (<https://epa.ms/App2-2>)

```
namespace PassXYZ.Vault;

public partial class App : Application ①
{
    public App()
    {
        InitializeComponent();
        MainPage = new AppShell(); ②
    }
}
```

In *Listing 2.2*, ① the `App` class is derived from the `Application` class, and the `Application` class is defined in the `Microsoft.Maui.Controls` namespace.

② `AppShell` is an instance of `Shell`, and it defines the UI of the initial page of the app. `Application` creates an instance of the `Window` class within which the application will run and views will be displayed. In the `App` class, we can overwrite the `CreateWindow` method to manage the lifecycle, which we will see soon.

Lifecycle management

In the .NET MAUI app, there are the following four lifecycle states:

- Running
- Not running
- Deactivated
- Stopped

During the state transitions, the predefined lifecycle events will be triggered. There are six cross-platform lifecycle events defined, as we can see in *Table 2.1*:

Event	Description	State transition	Override method
Created	This event is raised after the native window has been created.	Not running -> Running	OnCreated
Activated	This event is raised when the window has been activated and is, or will become, the focused window.	Not running -> Running	OnActivated
Deactivated	This event is raised when the window is no longer the focused window. However, the window might still be visible.	Running -> Deactivated	OnDeactivated
Stopped	This event is raised when the window is no longer visible.	Deactivated -> Stopped	OnStopped
Resumed	This event is raised when an app resumes after being stopped.	Stopped -> Running	OnResumed
Destroying	This event is raised when the native window is being destroyed and deallocated.	Stopped -> Not running	OnDestroying

Table 2.1: Lifecycle events and override methods

These lifecycle events are associated with the instance of the `Window` class created by `Application`. For each event, there is a corresponding override method defined. We can either subscribe to the lifecycle events or create override functions to handle lifecycle management.

Subscribing to the Window lifecycle events

To subscribe to the lifecycle events, as we can see in *Listing 2.3*, at ❶ we can override the `CreateWindow` method in the `App` class to create a `Window` instance on which we can subscribe to events:

Listing 2.3: App.xaml.cs with lifecycle events (<https://epa.ms/App2-3>)

```
using System.Diagnostics;  
namespace PassXYZ.Vault;
```

```
public partial class App : Application {
    public App() {
        InitializeComponent();
        MainPage = new MainPage();
    }

    protected override Window CreateWindow(IActivationState
        activationState) ❶
    {
        Window window = base.CreateWindow(activationState);
        window.Created += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 1. Created event");
        };
        window.Activated += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 2. Activated event");
        };
        window.Deactivated += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 3. Deactivated
                event");
        };
        window.Stopped += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 4. Stopped event");
        };
        window.Resumed += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 5. Resumed event");
        };
        window.Destroying += (s, e) => {
            Debug.WriteLine("PassXYZ.Vault.App: 6. Destroying
                event");
        };
        return window;
    }
}
```

In *Listing 2.3*, we revised the code of `App.xaml.cs` and we subscribed to all six events so that we can run a test and observe the state from the Visual Studio Output window, as shown next. After we launch our app, we can see that `Created` and `Activated` events are fired. Then, we minimize

our app. We can see that `Deactivated` and `Stopped` events are fired. When we resume our app, `Resumed` and `Activated` events are fired. Finally, we close our app, and a `Destroying` event is fired:

```
PassXYZ.Vault.App: 1. Created event
PassXYZ.Vault.App: 2. Activated event
PassXYZ.Vault.App: 4. Stopped event
PassXYZ.Vault.App: 3. Deactivated event
PassXYZ.Vault.App: 5. Resumed event
PassXYZ.Vault.App: 2. Activated event
PassXYZ.Vault.App: 5. Resumed event
PassXYZ.Vault.App: 2. Activated event
The thread 0x6f94 has exited with code 0 (0x0).
PassXYZ.Vault.App: 6. Destroying event
The program '[30628] PassXYZ.Vault.exe' has exited with code 0 (0x0).
```

Consuming the lifecycle override methods

Alternatively, we can consume the lifecycle override methods. We can create our own derived class from the `Window` class:

1. In Visual Studio, right-click on the project node and select **Add** and then **New Item...**
2. In the **Add New Item** window, select **C# Class** from the template and name it `PxWindow`. We created a new class, as shown next in *Listing 2.4*:

Listing 2.4: PxWindow.cs (<https://epa.ms/PxWindow2-4>)

```
using System.Diagnostics;
namespace PassXYZ.Vault;

public class PxWindow : Window
{
    public PxWindow() : base() {}
    public PxWindow(Page page) : base(page) {}

    protected override void OnCreated() {
        Debug.WriteLine("PassXYZ.Vault.App: 1. OnCreated");
    }
}
```

```
protected override void OnActivated() {
    Debug.WriteLine("PassXYZ.Vault.App: 2. OnActivated");
}
protected override void OnDeactivated() {
    Debug.WriteLine("PassXYZ.Vault.App: 3. OnDeactivated");
}
protected override void OnStopped() {
    Debug.WriteLine("PassXYZ.Vault.App: 4. OnStopped");
}
protected override void OnResumed() {
    Debug.WriteLine("PassXYZ.Vault.App: 5. OnResumed");
}
protected override void OnDestroying() {
    Debug.WriteLine("PassXYZ.Vault.App: 6. OnDestroying");
}
}
```

In *Listing 2.4*, we created a new class, `PxWindow`. In this class, we define our lifecycle override methods. We can use this new class in `App.xaml.cs`.

Next, let's look at the modified version of `App.xaml.cs` (*Listing 2.5*):

Listing 2.5 Modified App.xaml.cs with PxWindow (<https://epa.ms/App2-5>)

```
namespace PassXYZ.Vault;

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

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

1

When we repeat the test steps, we can see the following output from Visual Studio Output windows. The output looks very similar to the previous one. Basically, both approaches have the same effect on lifecycle management:

```
PassXYZ.Vault.App: 1. OnCreated
PassXYZ.Vault.App: 2. OnActivated
PassXYZ.Vault.App: 4. OnStopped
PassXYZ.Vault.App: 3. OnDeactivated
PassXYZ.Vault.App: 5. OnResumed
PassXYZ.Vault.App: 2. OnActivated
PassXYZ.Vault.App: 5. OnResumed
PassXYZ.Vault.App: 2. OnActivated
PassXYZ.Vault.App: 6. OnDestroying
The program '[25996] PassXYZ.Vault.exe' has exited with code 0
(0x0) .
```

We have learned about app lifecycle management in .NET MAUI through the Window class. We can either subscribe to lifecycle events or override the overridable methods to manage the app lifecycle. *Table 2.1* shows the comparison of these two approaches.

If you were a Xamarin.Forms developer, you might know that there were lifecycle methods defined in the Application class as well. In .NET MAUI, the following virtual methods are still available:

- OnStart – Called when the application starts
- OnSleep – Called each time the application goes to the background
- OnResume – Called when the application is resumed, after being sent to the background

To observe the behavior of these methods, we can override the following methods in our App class, as shown in *Listing 2.6*:

Listing 2.6: App.xaml.cs (<https://epa.ms/App2-6>)

```
using System.Diagnostics;

namespace PassXYZ.Vault;

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

```
        InitializeComponent();  
        MainPage = new MainPage();  
    }  
    protected override void OnStart() { ❶  
        Debug.WriteLine("PassXYZ.Vault.App: OnStart");  
    }  
    protected override void OnSleep() { ❷  
        Debug.WriteLine("PassXYZ.Vault.App: OnSleep");  
    }  
    protected override void OnResume() { ❸  
        Debug.WriteLine("PassXYZ.Vault.App: OnResume");  
    }  
}
```

When we test the preceding code on Windows, we can see the following debug message in the Visual Studio Output window:

```
PassXYZ.Vault.App: OnStart  
PassXYZ.Vault.App: OnSleep  
The thread 0x6844 has exited with code 0 (0x0).  
The thread 0x6828 has exited with code 0 (0x0).  
The thread 0x683c has exited with code 0 (0x0).  
PassXYZ.Vault.App: OnResume
```

As seen in *Listing 2.6*,

- ❶ When the app starts, we can see the `OnStart` method is invoked.
- ❷ When we minimize our app, we can see the `OnSleep` method is invoked.
- ❸ When we resume the app from the taskbar, the `OnResume` method is invoked.

We have learned about the lifecycle states of the .NET MAUI app. We also learned that we could subscribe to the lifecycle events or use override methods to manage the lifecycle. Next, let's explore the configuration of resources during the app startup.

Configuring the resources

Resource management is one of the major differences between .NET MAUI and Xamarin.

When it comes to cross-platform development, each platform has its own way of managing resources, and it's a daunting task for the development team to know and manage all those things. For example, we must include images in multiple sizes to support different resolutions.

In Xamarin, most of the resources are managed separately in platform-specific projects. If we want to add an image, we must add the image files with different sizes to all platform projects separately.

.NET MAUI provides an elegant solution to manage resources effectively. The design goal of one single project for all supported platforms helps to manage resources in one place.

In .NET MAUI, resource files can be tagged into different categories using a build action based on the role they play in the project, as we can see in *Table 2.2*:

Resource Type	Build Action	Example
Images	MauiImage	dotnet_bot.svg
Icons	MauiIcon	appicon.svg
Splash screen image	MauiSplashScreen	appiconfg.svg
Fonts	MauiFont	OpenSans-Regular.ttf
Style definition using external CSS	MauiCss	
Raw assets	MauiAsset	
XAML UI definition	MauiXaml	

Table 2.2: .NET MAUI resource types

After adding a resource file, the build action can be set in the **Properties** window in Visual Studio. If we look at the project file, we can see the following `ItemGroup`. If we put resources according to the convention of default folder setup, the resources will be treated as the respective category and the build action will be set automatically:

```
<ItemGroup>
  <!-- App Icon -->
  <MauiIcon Include="Resources\AppIcon\appicon.svg"
    ForegroundFile="Resources\AppIcon\appiconfg.svg"
    Color="#512BD4" />

  <!-- Splash Screen →
  <MauiSplashScreen Include="Resources\Splash\splash.svg"
```

```
        Color="#512BD4" BaseSize="128,128" />

    <!-- Images -->
    <MauiImage Include="Resources\Images\*" />

    <!-- Custom Fonts -->
    <MauiFont Include="Resources\Fonts\*" />
</ItemGroup>
```

App icon

In our app setup, we have an SVG image file, `appicon.svg`, under the `Resources\AppIcon` folder, with the build action set to `MauiIcon`. At build time, this file is used to generate the icon images on the target platform for various purposes, such as on the device, or in the app store.

It is possible to move this SVG file together with other images to the `Resources\Images` folder. In that case, we should use the following entry in the project file:

```
<MauiIcon Include="Resources\Images\appicon.png"
  ForegroundFile="Resources\Images\appiconfg.svg" Color="#512BD4"
/>
```

However, in this case, the build action is inconsistent for the files under the same folder, so `appicon.svg` resides in the `Resources\AppIcon` folder instead of `Resources\Images` in our project.

Splash screen

This is like an app icon. We have an SVG image file, `splash.svg`, in the `Resources/Splash` folder, with the build action set to `MauiSplashScreen`:

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

App icons, splash screens, and other images are simple resources, and we configure them in the project file directly.

Some frequently used resources, such as custom font and DI, may have to be configured in code, or both code and project files. We will discuss custom fonts here, and leave DI in *w, Dependency Injection and Refining Design*.

Setting custom font icons

Custom fonts can be managed as part of resources. In a mobile app, the visual representation is generally delivered through images. We use images in all kinds of navigation activities. In Android and iOS development, we need to manage image resources for different screen resolutions. In both Xamarin.Forms and .NET MAUI, we can use a custom font (icon font) instead of images for application icons.

In .NET MAUI, if controls can display text, these controls usually define properties that we can use to configure font settings for the text. The following properties are configurable:

- `FontAttributes`, which is an enumeration with three members: `None`, `Bold`, and `Italic`. The default value of this property is `None`.
- `FontSize`, which is the property of the font size, and the type is `double`.
- `FontFamily`, which is the property of the font family, and the type is `string`.

What is the advantage of using custom fonts for icons?

There are many advantages to using custom fonts as icons instead of images.

Font icons are vector icons instead of bitmap icons. Vector icons are scalable, meaning you don't need different images with different sizes and different resolutions based on the device. Icon font scaling can be handled through the `FontSize` property. The font file size is also much smaller than the images. A font file with hundreds of icons in it can be only a few KB in size.

Besides font size, the icon color can be changed with the `TextColor` property. With static images, we are not able to change the icon color.

Finally, font files can be managed in the shared project, so we don't have to manage fonts separately on different platforms.

Custom font setup

Custom font setup includes two parts – font files and configuration.

Custom font files can be added to the .NET MAUI or Xamarin.Forms shared project. The configuration of custom fonts in .NET MAUI is different than Xamarin.Forms. In Xamarin.Forms, we must configure it in `AssemblyInfo.cs`, while we can manage the configuration through .NET Generic Host in .NET MAUI.

In Xamarin.Forms, the process for accomplishing this is as follows:

1. Add the font file to the Xamarin.Forms shared project as an embedded resource (build action: `EmbeddedResource`).
2. Register the font file with the assembly in a file such as `AssemblyInfo.cs`, using the `ExportFont` attribute. An optional alias can also be specified.

In .NET MAUI, the process is changed and simplified as follows:

1. Add the font files in the Resources->Fonts folder. The build action is set to MauiFont, as we can see in *Figure 2.5*:

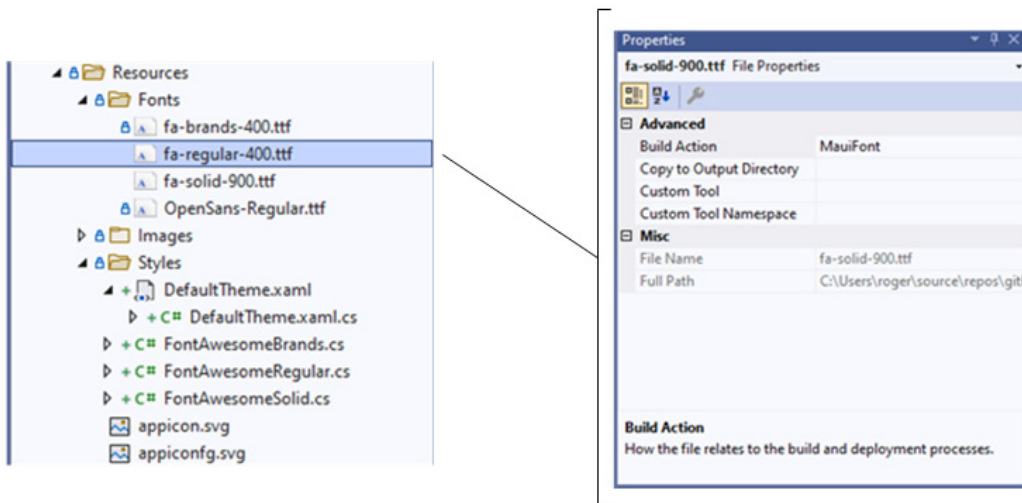


Figure 2.5: .NET MAUI Resources

2. Instead of registering the font file with the assembly, .NET MAUI initializes most of the resources through .NET Generic Host in the startup code, as shown in the following *Listing 2.7* at ❶. Font files are added using the `ConfigureFonts()` method, which is an extension method of the `MauiAppBuilder` class.

In our project, we use the Font Awesome icon library from the following open source project:

<https://github.com/FortAwesome/Font-Awesome>

The `fa-brands-400.ttf`, `fa-regular-400.ttf`, and `fa-solid-900.ttf` font files can be downloaded from the preceding website.

Let's review the source code *Listing 2.7* and see how to add these fonts to the app configuration.

Listing 2.7: MauiProgram.cs (<https://epa.ms/MauiProgram2-7>)

```
namespace PassXYZ.Vault;

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

```

{
    var builder = MauiApp.CreateBuilder();
    builder
        .UseMauiApp<App>()
        .ConfigureFonts(fonts => ❶
        {
            fonts.AddFont("fa-regular-400.ttf",
                "FontAwesomeRegular");
            fonts.AddFont("fa-solid-900.ttf", "FontAwesomeSolid");
            fonts.AddFont("fa-brands-400.ttf",
                "FontAwesomeBrands");
            fonts.AddFont("OpenSans-Regular.ttf",
                "OpenSansRegular");
        });
    return builder.Build();
}

```

In the above code, we can add fonts by invoking the `ConfigureFonts` **❶** method on the `MauiAppBuilder` object. To pass arguments to `ConfigureFonts`, we call `AddFont` method to add font to `IFontCollection` object.

Displaying font icons

To display font icons in .NET MAUI applications, we can specify the font icon data in a `FontImageSource` object. This class, which derives from the `ImageSource` class, has the following properties, as shown in *Table 2.3*:

Property name	Type	Description
Glyph	string	Unicode character value, such as " "
Size	double	The size of the font in device-independent units
FontFamily	string	A string representing the font family, such as <code>FontAwesomeRegular</code>
Color	Color	Font icon color in <code>Microsoft.Maui.Graphics.Color</code>

Table 2.3: Properties of `FontImageSource`

The following XAML example has a single font icon being displayed by an Image view:

```
<Image BackgroundColor="#D1D1D1">
    <Image.Source>
        <FontImageSource Glyph="&#xf007;"
                        FontFamily="FontAwesomeRegular"
                        Size="32" />
    </Image.Source>
</Image>
```

If you are not familiar with the XAML syntax in the preceding example, don't worry. We will learn about it in the next chapter. In the preceding code, a User icon is displayed in an Image control, which is from the FontAwesomeRegular font family that we just added in the configuration. The Glyph of the User icon is \uf007 in hex format and this is the C# escaped format. To use it in XML, we have to use the escaped format of XML, which is .

The equivalent C# code is as follows:

```
Image image = new Image {
    BackgroundColor = Color.FromHex("#D1D1D1")
};
image.Source = new FontImageSource {
    Glyph = "\uf007",
    FontFamily = "FontAwesomeRegular",
    Size = 32
};
```

In the preceding example, we refer to a font icon using Glyph in a hex number as a string. This is not convenient to use practically. We can define font glyphs as C# string constants so that we can refer to something more meaningful. There are many ways to do this. Here, we use the open source tool **IconFont2Code** to generate string constants. **IconFont2Code** can be found at the following URL in GitHub:

<https://github.com/andreinitescu/IconFont2Code>

We use Font Awesome in our project. On the website of IconFont2Code, we can upload our font library from the Resources\Fonts folder, and IconFont2Code will generate the code for us, as in the following example:

```
namespace PassXYZ.Vault.Resources.Styles;

static class FontAwesomeRegular
```

```

{
    public const string Heart = "\uf004";
    public const string Star = "\uf005";
    public const string Scan = "\uf006";
    public const string User = "\uf007";
    public const string Qrcode = "\uf008";
    public const string Fingerprint = "\uf009";
    public const string Clock = "\uf017";
    public const string ListAlt = "\uf022";
    public const string Flag = "\uf024";
    public const string Bookmark = "\uf02e";

    ...

    public const string SmileBeam = "\uf5b8";
    public const string Surprise = "\uf5c2";
    public const string Tired = "\uf5c8";
}

```

We can save the generated C# files in the Resources\Styles folder. The preceding file can be found here:

Resources\Styles\FontAwesomeRegular.cs

With the preceding FontAwesomeRegular static class, a font icon can be used just like the normal text in a XAML file:

```

<Button
    Text="Click me"
    FontAttributes="Bold"
    Grid.Row="3"
    SemanticProperties.Hint="Counts the number of times you
    click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center">
    <Button.ImageSource>
        <FontImageSource FontFamily="FontAwesomeSolid"
            Glyph="{x:Static app:FontAwesomeSolid.
                PlusCircle}"

```

```
        Color="{DynamicResource SecondaryColor}"
        Size="16" />
    </Button.ImageSource>
</Button>
```

In the preceding code, we add a circle plus icon to the `Button` control in front of the text "Click me". To refer to the icon name in the generated C# class, we added an app namespace, as defined here:

```
xmlns:app="clr-namespace:PassXYZ.Vault.Resources.Styles"
```

So far, we have created our project and configured the necessary resources that we need. It's time to build and test our app.

Building and debugging

As we recall in *Chapter 1, Getting Started with .NET MAUI*, regarding the development environment setup, we cannot build and test all targets using one platform. Please refer to *Table 1.8* about the available build targets on Windows and macOS platforms. To make it simple, we will build and test Windows and Android on the Windows platform. For iOS and macOS builds, we will do it on the macOS platform.

Once we are ready, we can build and debug our app.

Let's build and test on the Windows platform first. We can choose a framework that we want to run or debug, as shown in *Figure 2.6*:

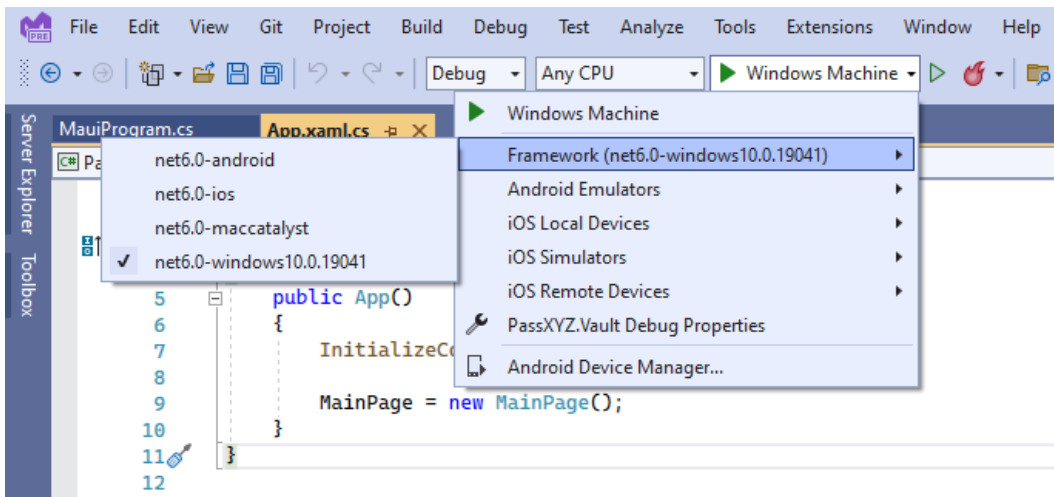


Figure 2.6: Building and debugging

Windows

We can run or debug a Windows build on a local machine by selecting `net6.0-windows10.0.19041` as the framework. To do this, we must enable Developer Mode on Windows, if it is not enabled yet. Please refer to *Figure 2.7* to enable Developer Mode on Windows 10 or 11:

1. Open the **Start** menu.
2. Search for **Developer settings** and select it.
3. Turn on **Developer Mode**.
4. If you receive a warning message about Developer Mode, read it, and select **Yes**.

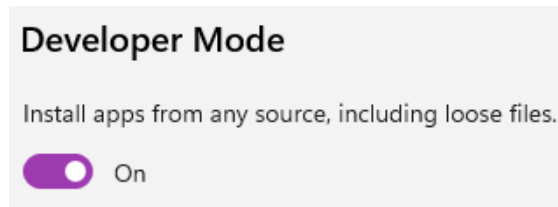


Figure 2.7: Developer Mode

Android

For the Android build, we can test it using an Android emulator or device. Before building or debugging, we need to connect a device or set up an instance of an emulator. Please refer to the following Microsoft documentation about how to set up a device or an emulator instance:

<https://learn.microsoft.com/en-us/dotnet/maui/>

We can run or debug from Visual Studio (*Figure 2.6*) by selecting `net6.0-android` as the framework.

Alternatively, we can also build and run from the command line using the following command:

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

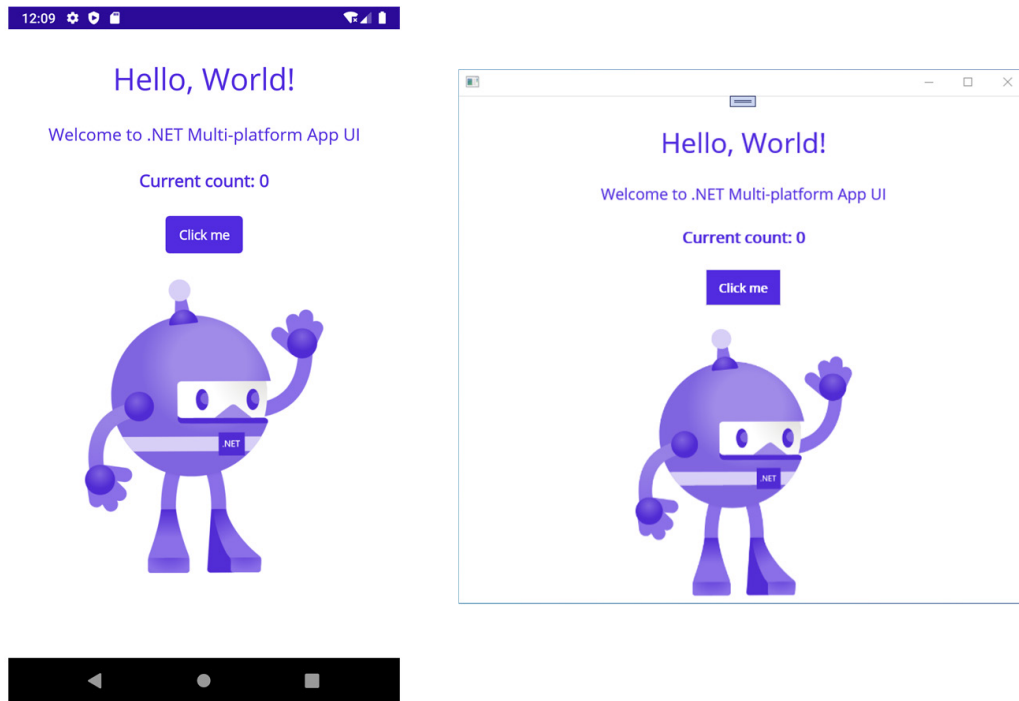


Figure 2.8: Running on Android and Windows

After we run the app on Android and Windows targets, we can see the preceding screen (*Figure 2.8*).

iOS and macOS

We can build and test iOS and macOS targets on a Mac computer. The steps to build and test using Microsoft Visual 2022 for Mac are similar to what we have done in Windows and Android. Let's look at how to do it using a command line.

To build and test the iOS target, we can use the following command in the project folder:

```
dotnet build -t:Run -f net6.0-ios -p:_  
DeviceName=:v2:udid=02C556DA-64B8-440B-8F06-F8C56BB7CC22
```

To select a target iOS emulator, we need to provide the device ID using the following parameter:

```
-p:_DeviceName=:v2:udid=
```

To find the device ID, we can launch Xcode on a Mac computer and go to **Windows -> Devices and Simulators**, as shown in *Figure 2.9*:

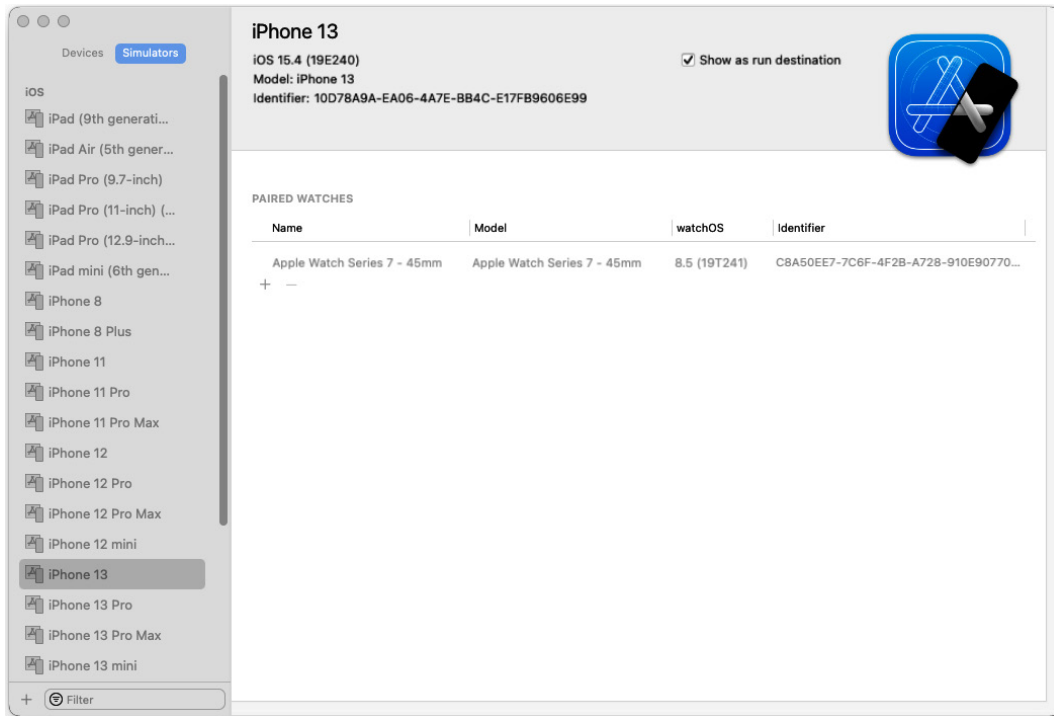


Figure 2.9: Devices and simulators in Xcode

For the macOS target, we can use the following command to build and test:

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

The screenshot for iOS and macOS is shown in *Figure 2.10* and we can see that the look and feel are similar to Android and Windows.

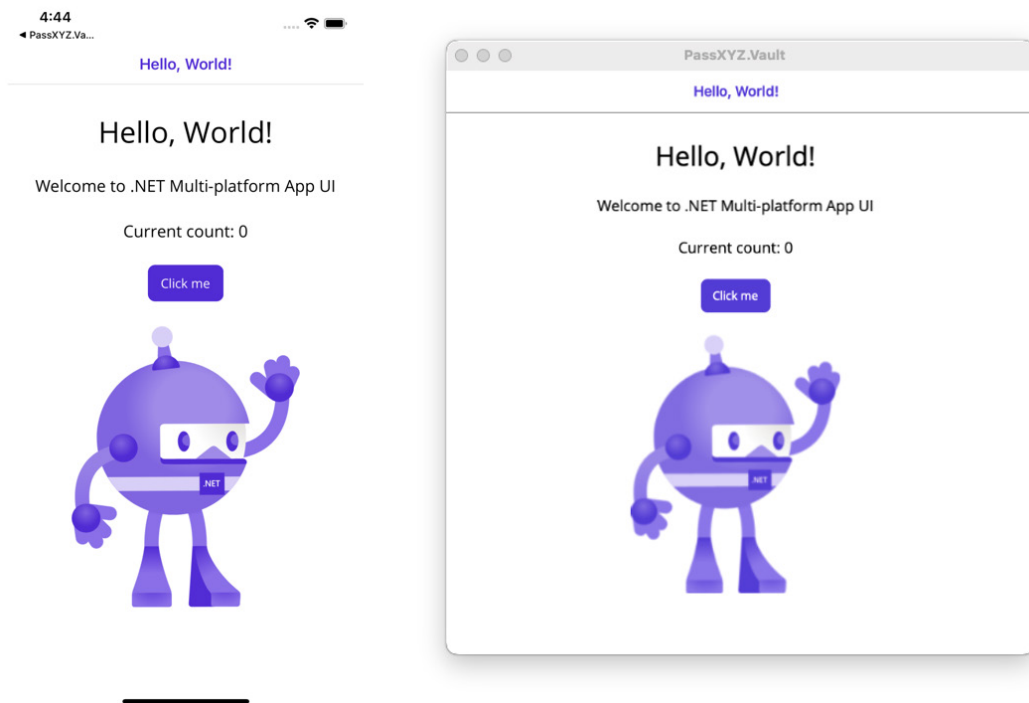


Figure 2.10: Running on iOS and macOS

The environment setup for Android, iOS, and macOS involves platform-specific details. Please refer to the Microsoft documentation for detailed instructions.

Even though this app works well, you can see that the app we just built is a simple one with only one window. To lay a better foundation for our subsequent development, we will use Shell as a navigation framework. There is a good Shell-based template in `Xamarin.Forms`, and we can use that to create the initial code for our app.

Scaffolding a Model-View-ViewModel project

We can run the app that we just created successfully now. We are going to develop a password manager app named *PassXYZ.Vault* in the rest of this book. Version *1.x.x* of this app is implemented in `Xamarin.Forms`, and you can find it in GitHub here:

<https://github.com/passxyz/Vault>

Version *1.x.x* is built using `Xamarin.Forms 5.0.0`. We will recreate it using `.NET MAUI` and share the experience in this book. The new release will be version *2.x.x*, and the source code is located here:

<https://github.com/passxyz/Vault2>

Shell is supported by `Microsoft.Maui.Controls.Shell` and `Xamarin.Forms.Shell` in .NET MAUI and `Xamarin.Forms`. It provides a common navigation user experience that can be used on all platforms. We will explain more about Shell in *Chapter 5, Introducing Shell and Navigation*. We will use .NET MAUI Shell as the UI framework and navigation method in this book.

The projects created from the Visual Studio template of both .NET MAUI and `Xamarin.Forms` use Shell. However, the default .NET MAUI project template contains only the simplest form of Shell, as we can see in *Listing 2.8*:

Listing 2.8: AppShell.xaml (<https://epa.ms/AppShell2-8>)

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

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

In *Listing 2.8*, if you are not familiar with XAML, don't worry. We will introduce XAML syntax in *Chapter 3, User Interface Design with XAML*. We can see that `MainPage` is displayed in `ShellContent`. It is the simplest UI without too much content in it. In our app, we will use the **Model-View-ViewModel (MVVM)** pattern and UI based on Shell. The MVVM pattern is a commonly used UI design pattern in .NET MAUI app development. We need to create the boilerplate code to include both the MVVM pattern and the Shell navigation structure. We could do it from scratch. However, the `Xamarin.Forms` template includes such boilerplate codes that I used in version *1.x.x* of *PassXYZ.Vault*. We can create the same project template for .NET MAUI. By doing this, we can also have an overview of how to migrate or reuse existing code from `Xamarin.Forms`.

Migrating and reusing a Shell template from Xamarin.Forms

Xamarin.Forms include a more complex Shell template that can be configured to generate boilerplate code for flyout or tabbed Shell navigation. We can create a new Xamarin.Forms project using this template. After that, we can use this boilerplate code in the .NET MAUI app that we just created.

To create a new Xamarin.Forms project, follow these steps:

1. Launch Visual Studio 2022 and select **Create a new project**. This opens the **Create a new project** wizard. In the search box, we can type `Xamarin`, and all Xamarin-related project templates will be shown (see *Figure 2.11*):

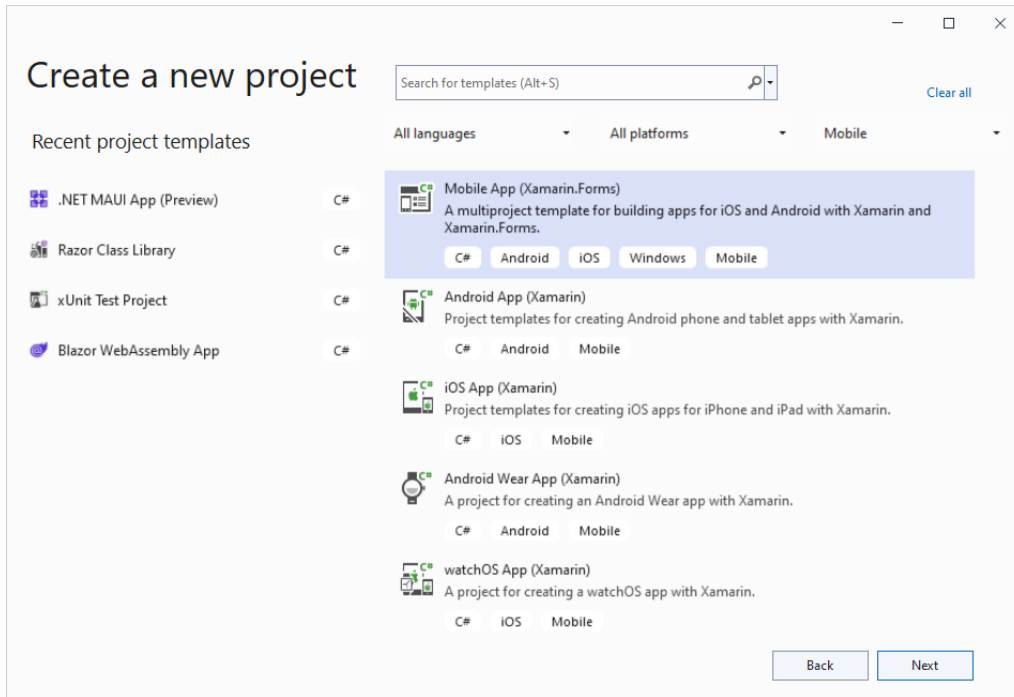


Figure 2.11: New Xamarin project

2. Select **Mobile App (Xamarin.Forms)** from the list and click **Next**. On the next screen, as shown in *Figure 2.12*, we should choose a different location, but use the same project name, `PassXYZ.Vault`, and click the **Create** button:

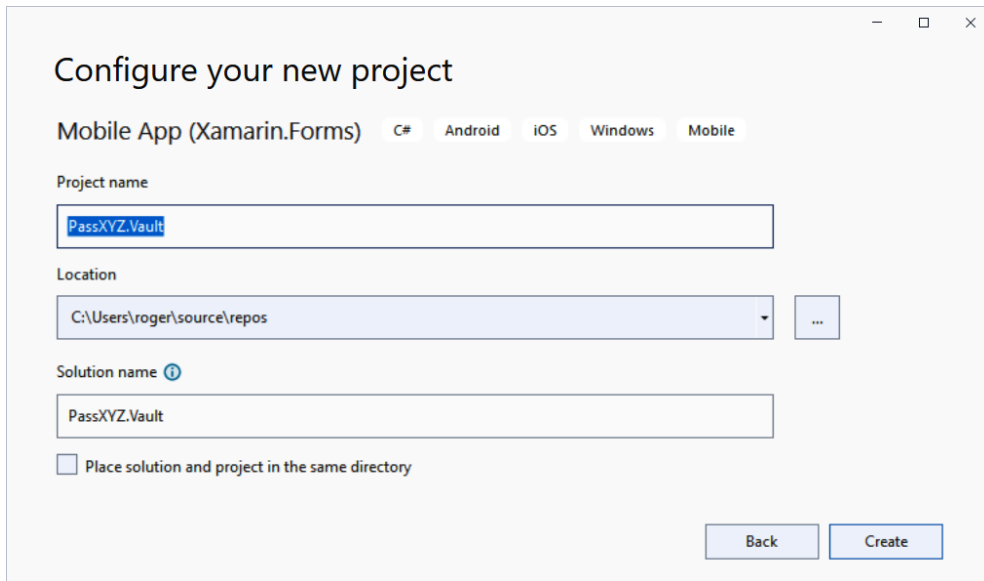


Figure 2.12: Configure the Xamarin project

3. We have one more step, as shown in *Figure 2.13*. Let's select the **Flyout** template and click **Create**:

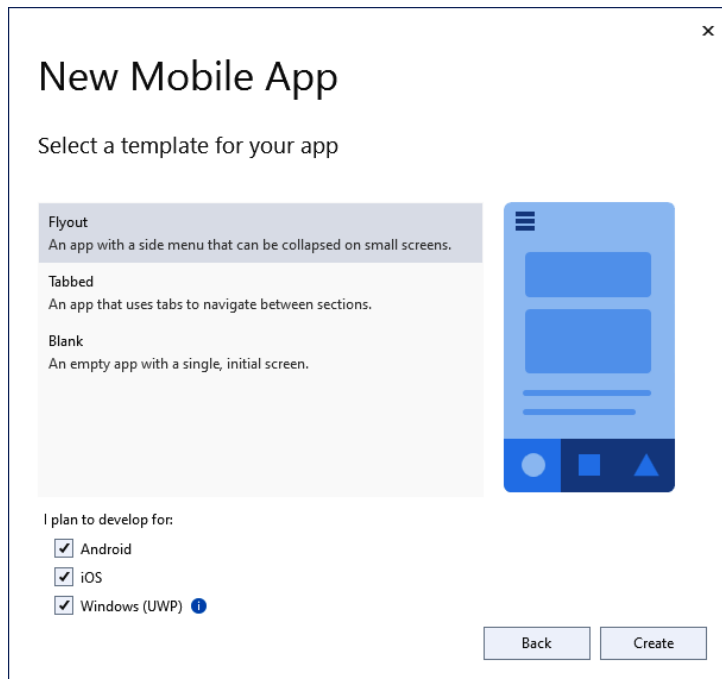


Figure 2.13: Configure the Xamarin project – Flyout

After the new solution is created, we can see that there are four projects in the solution, as shown in *Figure 2.14*:

- **PassXYZ.Vault** – This is a .NET Standard project that is shared by other projects, and all platform-independent code should be here
- **PassXYZ.Vault.Android** – This is the Android platform-specific project
- **PassXYZ.Vault.iOS** – This is the iOS platform-specific project
- **PassXYZ.Vault.UWP** – This is the UWP-specific project

We can see that the project structure of Xamarin.Forms is quite different from .NET MAUI. There are multiple projects in the solution. All resources are managed separately in platform-specific projects. Most of the development work should be done in the .NET Standard project, *PassXYZ.Vault*, and we will migrate and reuse the code in this project.

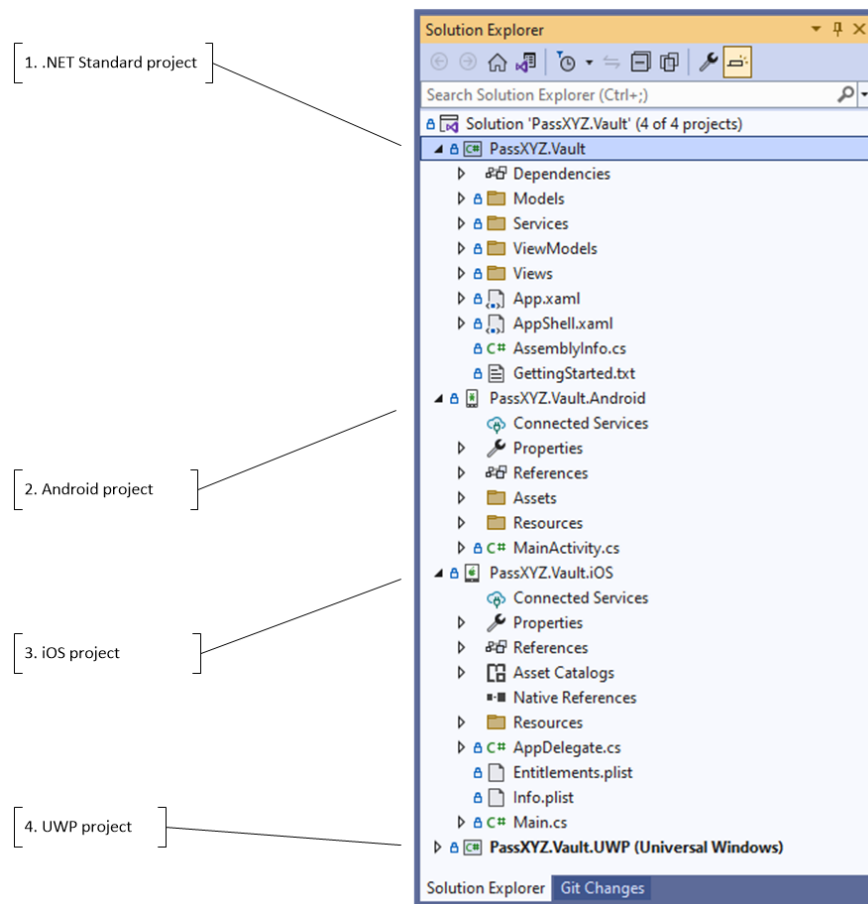


Figure 2.14: Xamarin.Forms project structure

The source code of this Xamarin.Forms project can be found here:

<https://github.com/shugaoye/PassXYZ.Vault2/tree/xamarin>

If platform-specific code is not involved, the migration process is relatively simple. We are handling the simplest case here. The production code is usually much more complicated than this so any migration should be planned after a detailed analysis.

Let's focus on the .NET Standard project. The content in the .NET Standard project includes the boilerplate code of the MVVM pattern and Shell UI, which is what we need. We can copy files in the previous list to the .NET MAUI project and change namespaces in the source code.

The following are the steps to be taken in the migration process:

1. Please refer to *Table 2.4*, which shows a list of actions corresponding to the list of files and folders in the .NET Standard project:

Xamarin.Forms	Actions	.NET MAUI
App.xaml	No	Keep the .NET MAUI version. It defines the instance of the <code>Application</code> class.
AppShell.xaml	Replace	Overwrite the .NET MAUI version and change namespaces to .NET MAUI. This file defines the Shell navigation hierarchy.
Views/	Copy	New folder in .NET MAUI project. Need to change namespaces.
ViewModels/	Copy	New folder in .NET MAUI project. Need to change namespaces.
Services/	Copy	Interface to export models. New folder in .NET MAUI project. Need to change namespaces.
Models/	Copy	New folder in .NET MAUI project. Need to change namespaces.

Table 2.4: Actions in the .NET standard project

2. In the .NET MAUI project, please refer to *Table 2.4* to replace the following namespaces:

Old namespace	New namespace
<code>xmlns="http://xamarin.com/schemas/2014/forms"</code>	<code>xmlns="http://schemas.microsoft.com/dotnet/2021/maui"</code>
<code>using Xamarin.Forms</code>	<code>using Microsoft.Maui AND using Microsoft.Maui.Controls</code>
<code>using Xamarin.Forms.Xaml</code>	<code>using Microsoft.Maui.Controls.Xaml</code>

Table 2.5: Namespaces in .NET MAUI and Xamarin.Forms

3. Test and fix any errors.

In *Figure 2.15*, we can see that the list of files changed is views and view models:

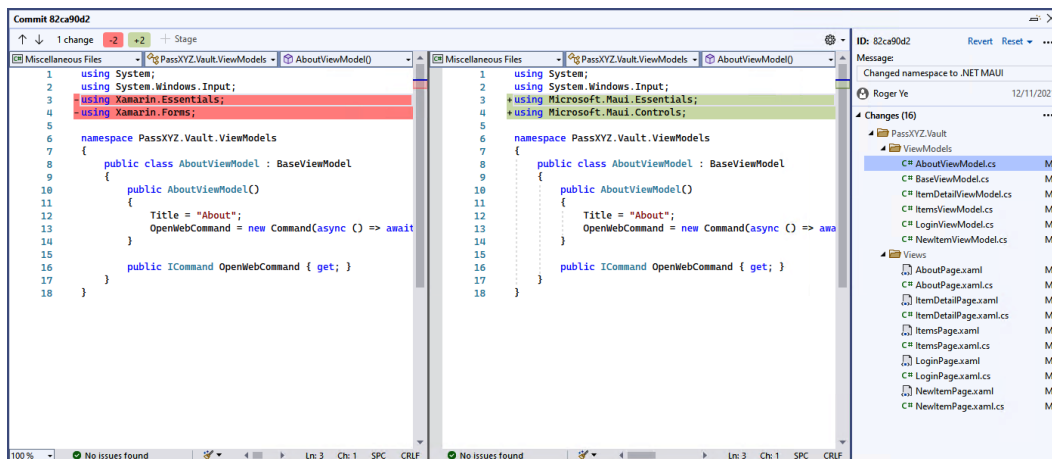


Figure 2.15: Changed files in migration (<https://bit.ly/3N1fqvO>)

For this simple case, all changes are related to the namespace. This is not true in real-world situations. Even though the process looks simple, it is still relatively complicated for people who are new to .NET MAUI. I don't recommend you to repeat this. Instead, I created a new Visual Studio project template to include the desired outcome.

Let's build and test this updated app.

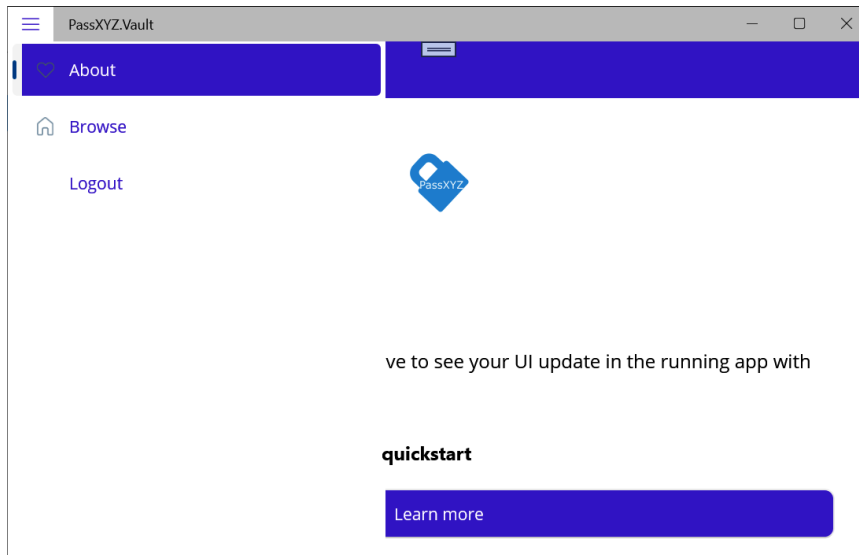


Figure 2.16: PassXYZ.Vault with .NET MAUI Shell

In *Figure 2.16*, we can see that there are three pages included in the default Shell menu:

- **About** – This is an about page
- **Browse** – This is the entry point of a list of items
- **Logout** – This is the link to the login page where you can log in or log out

We will use this as the boilerplate code for further development in this book. To summarize the work that we have done in this section, I created a Visual Studio project template for it. By using this project template, we can generate the project structure we want.

Visual Studio project template

The project template can be downloaded as a Visual Studio extension package from the Visual Studio Marketplace, as shown in *Figure 2.17*:

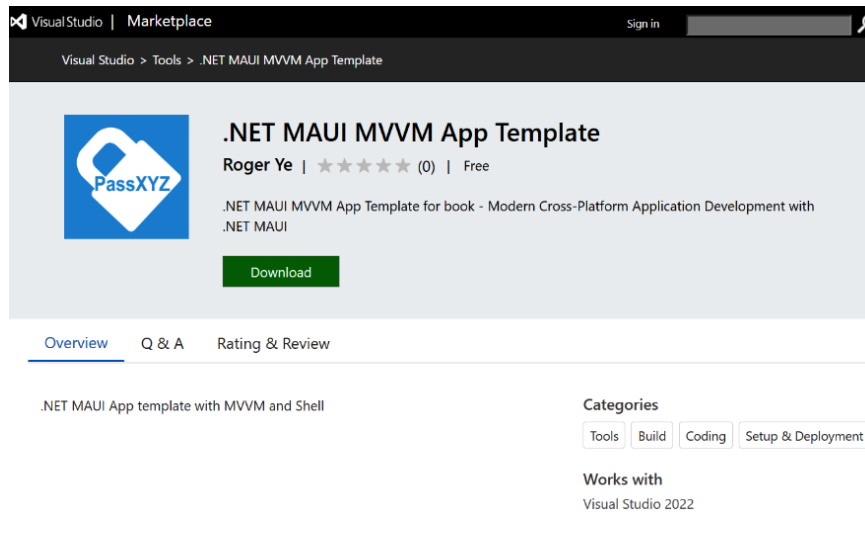


Figure 2.17: Project template in Visual Studio Marketplace

After the installation of this project template, we can create a new .NET MAUI project, as shown in Figure 2.18:

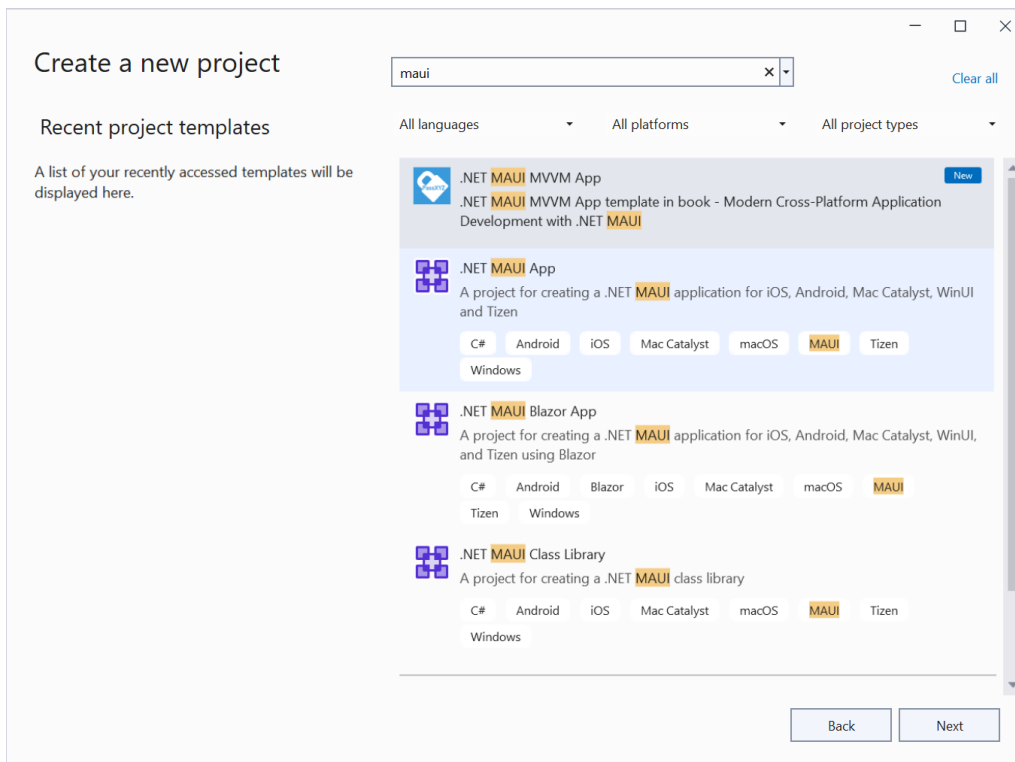


Figure 2.18: Creating a new .NET MAUI project using the project template

In the project created using this template, the project structure is the same as the one in this chapter. The source code of this project template can be found here:

<https://github.com/passxyz/MauiTemplate>

Summary

We created a new .NET MAUI project in this chapter. We learned how to configure the .NET MAUI app using .NET Generic Host, and we can use a custom font (Font Awesome) after updating the configuration of the resources. We also learned about the .NET MAUI application lifecycle. We tested how to subscribe to lifecycle events by overriding the `CreateWindow` method and by creating a derived class of the `Window` class. To create the boilerplate code with the MVVM pattern and Shell support, we created a new .NET MAUI project template. Throughout the process, we demonstrated how to migrate Xamarin.Forms code to .NET MAUI.

In the next chapter, we will learn how to create a user interface using XAML. XAML can be used to build user interfaces for WPF, UWP, Xamarin.Forms, and .NET MAUI. We will create and improve the user interfaces of our password manager app using XAML.

3

User Interface Design with XAML

We created a new .NET MAUI project called PassXYZ.Vault in the previous chapter. We will start to improve it with the capabilities that we will master throughout this book. In this chapter, we will use the master-detail pattern and XAML to design and build our app user interface.

The following topics will be covered in this chapter related to user interface design with XAML:

- How to create a XAML page
- Basic XAML syntax
- XAML markup extension
- How to design the user interface with the master-detail pattern
- Localization of the .NET MAUI app

The **eXtensible Application Markup Language (XAML)** is an XML-based language that is used to define user interfaces for **Windows Presentation Foundation (WPF)**, **Universal Windows Platform (UWP)**, Xamarin.Forms, and .NET MAUI. The XAML dialects in these platforms share the same syntax but differ in their vocabularies.

XAML allows developers to define user interfaces in XML-based markup language rather than code. It is possible to write all our user interfaces in code, but the user interface design with XAML will be more succinct and more visually coherent. XAML cannot contain code. This is a disadvantage, but it is also an advantage as it forces the developer to separate the code logic from the user interface design. XAML is well suited for the **Model-View-ViewModel (MVVM)** pattern, which we will learn about later in this book.

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code for this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter03>

Creating a XAML page

Before we learn XAML syntax, let's learn how to create a XAML page in Visual Studio and via the `dotnet` command line.

To create a XAML page using Visual Studio, we can right-click on the project node. After, select **Add > New Item...**; we will see what's shown in *Figure 3.1*:

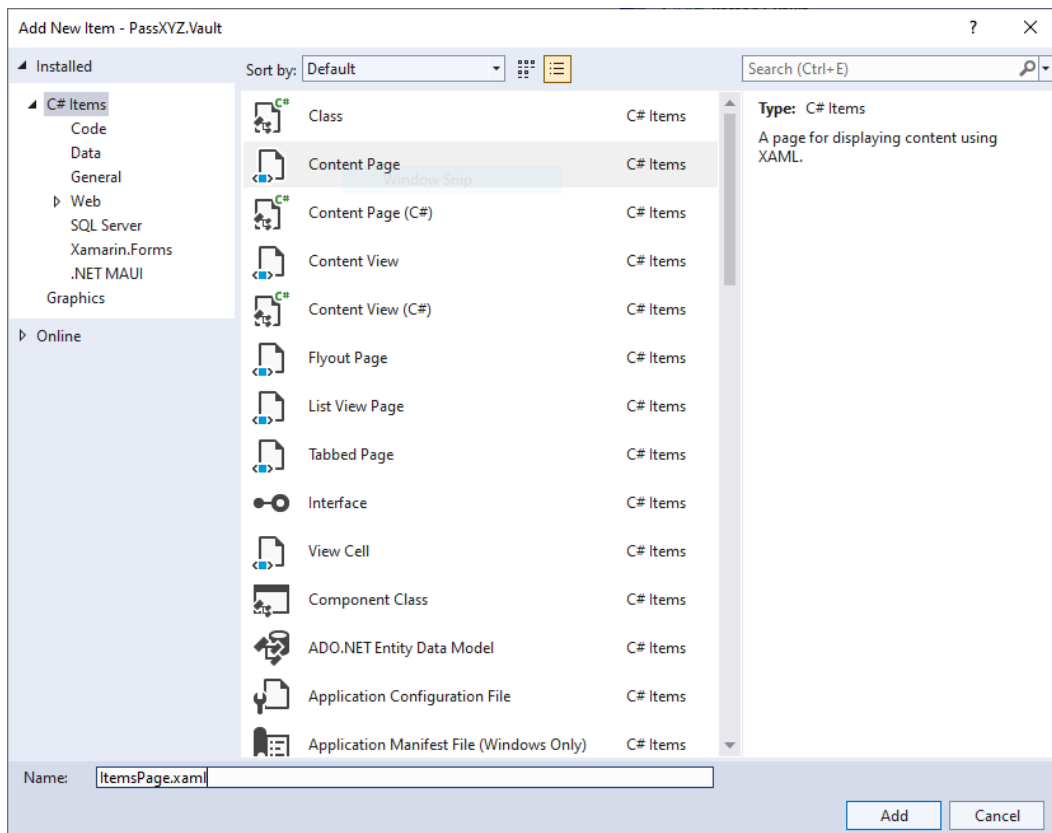


Figure 3.1: Adding a XAML page

On this screen, select **Content Page** from the templates and click **Add**. This will create a pair of files – a XAML file and a C# code-behind file.

We can do the same using the `dotnet` command.

To find all .NET MAUI templates, we can use the `dotnet` command like so in a PowerShell console:

```
dotnet new --list | findstr -i maui

.NET MAUI App                maui                [C#]  MAUI/...
.NET MAUI Blazor App         maui-blazor         [C#]  MAUI/...
.NET MAUI Class Library      mauilib             [C#]  MAUI/...
.NET MAUI ContentPage (C#)   maui-page-csharp    [C#]  MAUI/...
.NET MAUI ContentPage (XAML) maui-page-xaml      [C#]  MAUI/...
.NET MAUI ContentView (C#)   maui-view-csharp    [C#]  MAUI/...
.NET MAUI ContentView (XAML) maui-view-xaml      [C#]  MAUI/...
.NET MAUI ResourceDictionary (XAML) maui-dict-xaml     [C#]  MAUI/...
```

From the preceding output, we can see that the short name of the XAML content page is `maui-page-xaml`. We can create a XAML page using the following command:

```
dotnet new maui-page-xaml -n ItemsPage

The template ".NET MAUI ContentPage (XAML)" was created
successfully.

Processing post-creation actions...
The post action 84c0da21-51c8-4541-9940-6ca19af04ee6 is not
supported.
Description: Opens NewPage1.xaml in the editor.
```

Two files called `ItemsPage.xaml` and `ItemsPage.xaml.cs` will be created by the preceding command.

XAML syntax

Since XAML is an XML-based language, we need to understand basic XML syntax. An XML or XAML file includes a hierarchy of elements. Each element may have attributes associated with it. Let's review `App.xaml` in the project that we created in *Chapter 2, Building Our First .NET MAUI App*, as an example:

Listing 3.1: `App.xaml` (<https://epa.ms/App3-1>)

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

As shown in *Listing 3.1*, we have an element called `Application` and its associated attributes, `x:Class` and `xmlns:{prefix}`. Let's analyze this example to understand XML elements and attributes.

Element

The element syntax includes a start and end tag, such as the `Application` tag:

```
<Application></Application>
```

For an empty element, the end tag can be omitted by adding a forward slash at the end of the start tag, like so:

```
<Application />
```

When we mention an XML element, we may use `element`, `node`, and `tag` as terms. When we say `element`, we are referring to the start tag and the end tag of that element together. When we say `tag`, we are referring to either the start or end tag of the element. When we say `node`, we are referring to an element and all its inner content, including all child elements.

A XAML document is comprised of many nested elements. There is only one top element, which is called the root element. In .NET MAUI or Xamarin.Forms, the root element is usually `Application`, `ContentPage`, `Shell`, or `ResourceDictionary`.

For each XAML file, we usually have a corresponding C# code-behind file. Let's review the code-behind file in *Listing 3.2*:

Listing 3.2: App.xaml.cs (<https://epa.ms/App3-2>)

```
using PassXYZ.Vault.Services;
using PassXYZ.Vault.Views;
namespace PassXYZ.Vault;

public partial class App : Application { ❶
    public App() {
        InitializeComponent();           ❷
        Routing.RegisterRoute(nameof(ItemDetailPage),
                               typeof(ItemDetailPage));
        Routing.RegisterRoute(nameof(NewItemPage),
                               typeof(NewItemPage));
        DependencyService.Register<MockDataStore>();
        MainPage = new AppShell();
    }
    private async void OnMenuItemClicked(System.Object
        sender, System.EventArgs e) {
        await Shell.Current.GoToAsync("//LoginPage");
    }
}
```

In XAML, elements usually represent actual C# classes that are instantiated to objects at runtime. Together, the XAML and code-behind files define a complete class. For example, `App.xaml` (*Listing 3.1*) and `App.xaml.cs` (*Listing 3.2*) define the `App` class, which is a sub-class of `Application`.

❶ The `App` class, whose full name is `PassXYZ.Vault.App`, is the same as the one defined in the XAML file using the `x:Class` attribute:

```
x:Class="PassXYZ.Vault.App"
```

❷ In the constructor of the App class, the `InitializeComponent()` method is called to load the XAML and parse it. UI elements defined in the XAML file are created at this point. We can access these UI elements by the name defined with the `x:Name` attribute, as we'll see shortly.

Attribute

An element can have multiple unique attributes. An attribute provides additional information about XML elements. An XML attribute is a name-value pair attached to an element. In XAML, an element represents a C# class and attributes represent the members of this class:

```
<Button x:Name="loginButton" VerticalOptions="Center"
        IsEnabled="True" Text="Login"/>
```

As we can see, four attributes – `x:Name`, `VerticalOptions`, `IsEnabled`, and `Text` – are defined for the `Button` element. To define an attribute, we need to specify the attribute's name and value with an equal sign. We need to put the attribute value in double or single quotes. For example, `IsEnabled` is the attribute name and `"True"` is the attribute value.

In this example, the `x:Name` attribute is a special one. It does not refer to a member of the `Button` class, but it refers to the variable holding the instance of the `Button` class. Without the `x:Name` attribute, an anonymous instance of the `Button` class will be created. With the `x:Name` attribute declared, we can refer to the instance of the `Button` class using the `loginButton` variable in the code-behind file.

XML namespaces and XAML namespaces

In XML or XAML, we can declare namespaces just like we do in C#. Namespaces help to group elements and attributes to avoid name conflicts when the same name is used in a different scope. Namespaces can be defined using the `xmlns` attribute with the following syntax:

```
xmlns:prefix="identifier"
```

The XAML namespace definition has two components: a prefix and an identifier. Both the prefix and the identifier can be any string, as allowed by the W3C namespaces in the XML 1.0 specification. If the prefix is omitted, the namespace is the default namespace. In *Listing 3.1*, the following namespace is the default one:

```
xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
```

This default namespace allows us to refer to .NET MAUI classes without a prefix, such as `ContentPage`, `Label`, or `Button`.

For the namespace declaration, use the `x` prefix, like so:

```
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

The `xmlns:x` namespace declaration specifies elements and attributes that are intrinsic to XAML. This namespace is one of the most important ones that we will use in the UI design with XAML. To understand how to use it, we can create a content page with the same structure using both C# and XAML.

To create a content page in XAML, we can use the `dotnet` command, as we did previously:

```
dotnet new maui-page-xaml -n NewPage1
The template ".NET MAUI ContentPage (XAML)" was created
successfully.
Processing post-creation actions...
The post action 84c0da21-51c8-4541-9940-6ca19af04ee6 is not
supported.
Description: Opens NewPage1.xaml in the editor.
```

The preceding command generates a XAML file (`NewPage1.xaml`) and a C# code-behind file (`NewPage1.xaml.cs`). We can update the XAML file to the following. Since we aren't adding any logic, we can ignore the code-behind file (`NewPage1.xaml.cs`) in this example:

NewPage1.xaml

```
<ContentPage
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="MauiApp1.NewPage1"
  Title="NewPage1">
  <StackLayout x:Name="layout">
    <Label Text="Welcome to .NET MAUI!"
      VerticalOptions="Center"
      HorizontalOptions="Center" />
  <BoxView HeightRequest="150" WidthRequest="150"
    HorizontalOptions="Center">
    <BoxView.Color>
      <Color x:FactoryMethod="FromRgba">
        <x:Arguments>
          <x:Int32>192</x:Int32>
          <x:Int32>75</x:Int32>
```

```
        <x:Int32>150</x:Int32>
        <x:Int32>128</x:Int32>
    </x:Arguments>
</Color>
</BoxView.Color>
</BoxView>
</StackLayout>
</ContentPage>
```

NewPage1.xaml.cs

```
namespace MauiApp1;
public partial class NewPage1 : ContentPage {
    public NewPage1() {
        InitializeComponent();
    }
}
```

We can also generate a content page in C# code only. Let's create a content page using the following command:

```
dotnet new maui-page-csharp -n NewPage1
The template ".NET MAUI ContentPage (C#)" was created
successfully.

Processing post-creation actions...
The post action 84c0da21-51c8-4541-9940-6ca19af04ee6 is not
supported.
Description: Opens NewPage1.cs in the editor.
```

The preceding command generates a content page in the `NewPage1.cs` C# file. We can implement the same logic in C# like so:

NewPage1.cs

```
namespace MauiApp1;
public class NewPage1 : ContentPage {
    public NewPage1() {
        var layout = new StackLayout
    }
```

①

②

```

    {
        Children = {
            new Label { Text = "Welcome to .NET MAUI!" },
            new BoxView {
                HeightRequest = 150,
                WidthRequest = 150,
                HorizontalOptions = LayoutOptions.Center,
                Color = Color.FromRgba(192, 75, 150, 128) ③
            }
        }
    };
    Content = layout;
}
}

```

Here, we have created the same content page (`NewPage1`) twice in both XAML and C#. XAML cannot contain programming logic, but it can be used to declare user interface elements and put the logic in the C# code-behind file. Inside `NewPage1`, we created a content page that contains `Label` and `BoxView` elements. In the XAML version, we used attributes defined in the `xmlns:x` namespace to specify the UI elements:

- ❶ A content page called `NewPage1` is created in XAML. The `x:Class` attribute specifies the class name – that is, `NewPage1`. In the C# code-behind file, a partial class of `NewPage1` is defined. In the constructor, the `InitializeComponent()` method is invoked to load the UI defined in XAML.
- ❷ We can create the same content page, `NewPage1`, using C# directly as a derived class of `ContentPage`.

We defined a `StackLayout` in the content page and the variable name referring to it is `layout` in both the XAML and C# versions:

- ❷ In XAML, `x:Name` specifies the variable name of `StackLayout`.
- ❷ In C#, we can declare the variable as `layout`.
- ❸ `x:FactoryMethod` specifies a factory method that can be used to initialize an object.
- ❸ In C# code, we can call the `Color.FromRgba()` function directly, but we have to use the `x:FactoryMethod` attribute in XAML to do the same.
- ❹ `x:Arguments` is used to specify arguments when we call `Color.FromRgba()` in XAML.
- ❺ `x:Int` is used to specify integer arguments. For other data types, we can use `x:Double`, `x:Char`, or `x:Boolean`.

For more information about the `xmlns:x` namespace, please refer to the Microsoft documentation at <https://learn.microsoft.com/en-us/dotnet/maui/xaml/namespaces/>.

Common Language Runtime (CLR) types can be referenced in XAML by declaring a XAML namespace with a prefix. As shown in *Listing 3.1*, we can refer to our C# namespace, `PassXYZ.Vault`, like so:

```
xmlns:local="clr-namespace:PassXYZ.Vault"
```

To declare a CLR namespace, we can use `clr-namespace:` or `using::`. If the CLR namespace is defined in a different assembly, `assembly=` is used to specify the assembly that contains the referenced CLR namespace. The value is the name of the assembly without the file extension. In our case, it has been omitted since the `PassXYZ.Vault` namespace is within the same assembly as our application code.

We will see more uses of namespaces later in this chapter.

XAML markup extensions

Even though we can initialize class instances using XAML elements and set class members using XAML attributes, we can only set them as predefined constants in a XAML document.

To enhance the power and flexibility of XAML by allowing element attributes to be set from a variety of sources, we can use XAML markup extensions. With XAML markup extensions, we can set an attribute to values defined somewhere else, or a result processed by code at runtime.

XAML markup extensions can be specified in curly braces, as shown here:

```
<Button Margin="0,10,0,0" Text="Learn more"
        Command="{Binding OpenWebCommand}"
        BackgroundColor="{DynamicResource PrimaryColor}"
        TextColor="White" />
```

In the preceding code, both the `BackgroundColor` and `Command` attributes have been set to markup extensions. `BackgroundColor` has been set to `DynamicResource` and `Command` has been set to the `OpenWebCommand` method defined in the view model.

We will use markup extensions in the next few chapters to support data binding and `ResourceDictionary`. We will learn more about markup extensions when we use them later. Please refer to the following Microsoft documentation to find out more information about it: <https://learn.microsoft.com/en-us/dotnet/maui/xaml/markup-extensions/consume>.

Now that we've learned the basics about XAML, we can use it to work on our user interface design.

Master-detail UI design

We have learned basic knowledge about XAML in the previous sections. Now, let's spend some time exploring the application that we are going to develop.

The master-detail pattern is commonly used in user interface design. Many examples can be found in frequently used apps. For example, in the Mail app of Windows, a list of emails is displayed in the master view, as well as the details of the selected email:

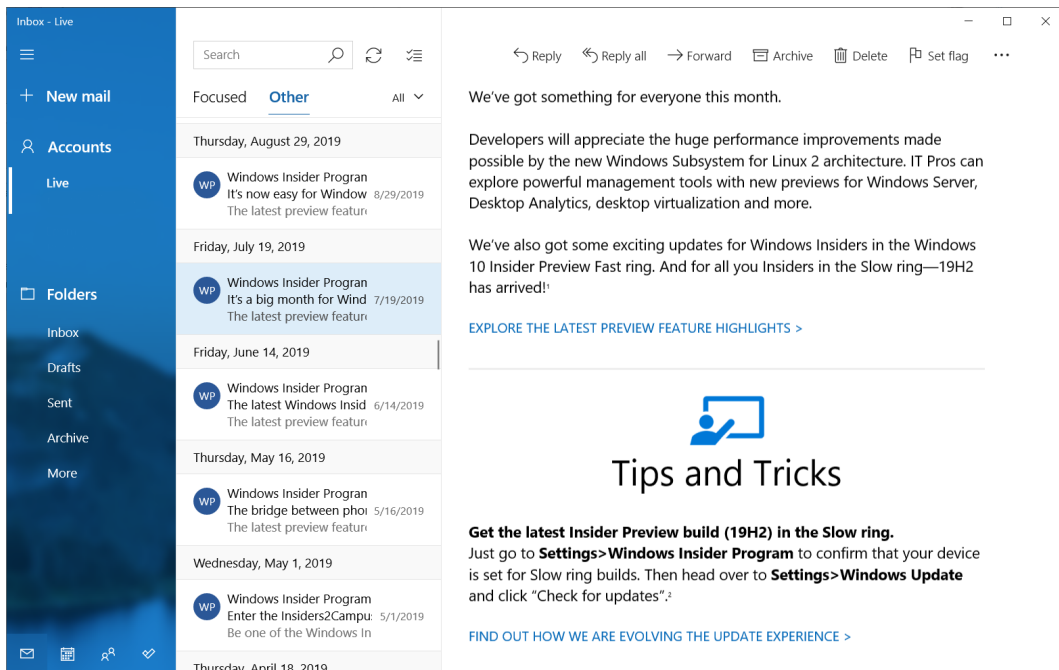


Figure 3.2: Mail in Windows

In *Figure 3.2*, there are three panels in the design. The left panel looks like a navigation drawer. When we select a folder from the left panel, a list of emails is displayed in the middle panel. The currently selected email is displayed in the right panel.

Note

Navigation drawers provide access to destinations and app functionality, such as the menu in the desktop environment. It typically slides in from the left and is triggered by tapping an icon in the top-left corner of the screen. It displays a list of choices to navigate to and is widely used in mobile and web user interface design. Xamarin.Forms or .NET MAUI Shell uses navigation drawers as their top-level navigation methods.

The original KeePass UI design, shown in *Figure 3.3*, also uses three panels (left, right, and bottom) on the main page. The left panel is a classic tree view that acts like a navigation drawer. The right panel is used to display the list of password entries. The bottom panel is used to display the details of an entry:

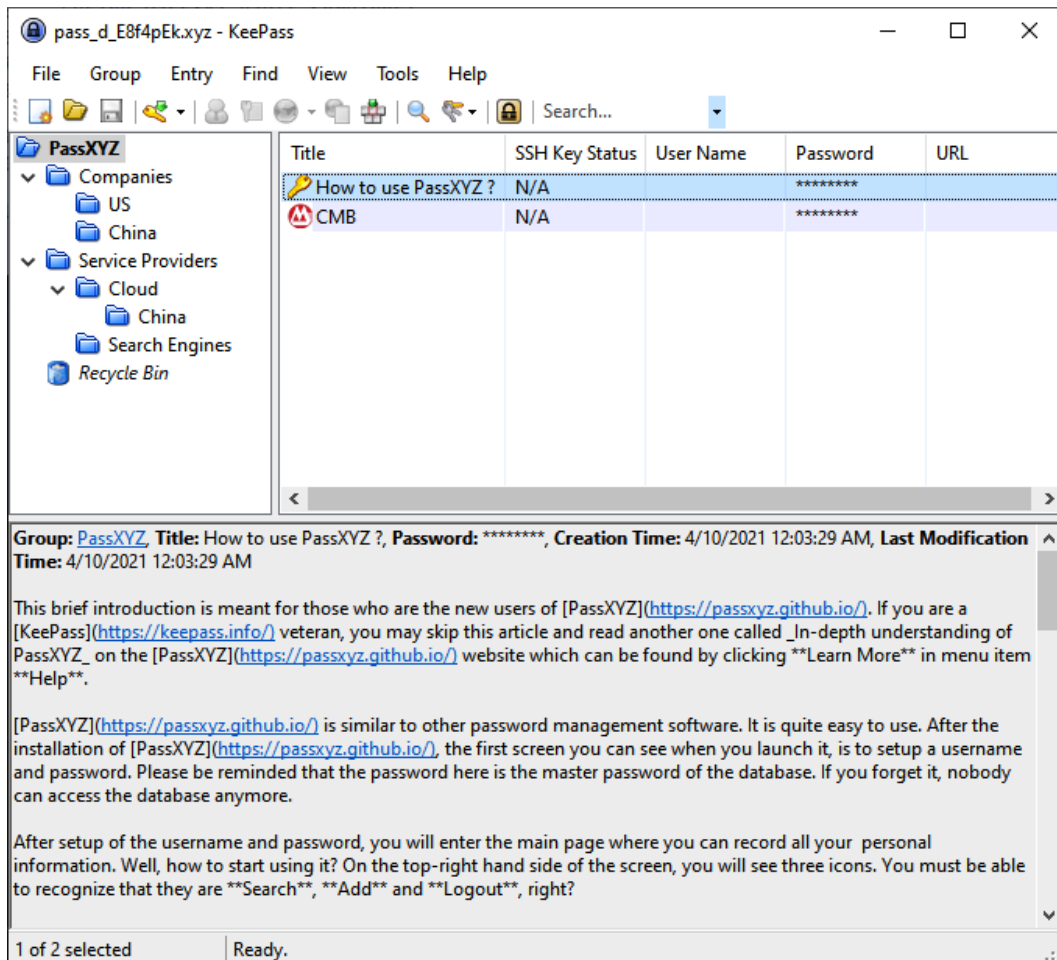


Figure 3.3: KeePass UI design

The master-detail pattern works well on a wide range of device types and display sizes.

Considering different display sizes, two popular modes can be used:

- Side-by-side
- Stacked

Side-by-side

When we have plenty of horizontal space with a large display, the side-by-side approach is usually a good choice. The Mail app in *Figure 3.2* and the KeePass app in *Figure 3.3* are good examples. In this mode, we can see both the master view and the detail view at the same time.

Stacked

When we use a mobile device, we usually have a smaller screen size and the vertical space is larger than the horizontal one. The stacked approach is a better choice in this case.

In stacked mode, the master view gets the full-screen space. Then, when a selection is made, the detail view gets the full-screen space:

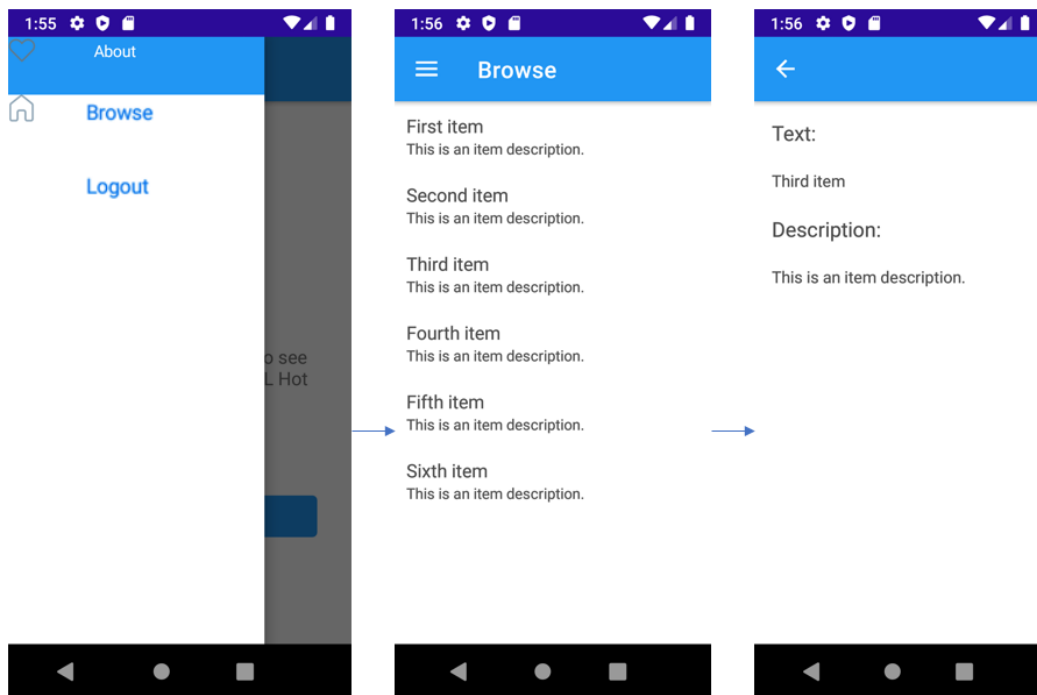


Figure 3.4: PassXYZ.Vault

In *Figure 3.4*, we can see how we can navigate our app from the user's point of view. We have a list of flyout items that we can choose from:

- **About**
- **Browse**
- **Logout**

When we choose **Browse**, we can see the list of items on the master page (`ItemsPage`). On this page, if we choose an item, we will go to the item's detail page (`ItemDetailPage`). If we want to choose another item, we must go back to the master page and select another item.

We will discuss flyout items in *Chapter 5, Introducing Shell and Navigation*. In this section, we will review the implementation of `ItemsPage` and `ItemDetailPage`. However, before we go into the details, let's study the layout, which is the container of user interface elements.

Controls in .NET MAUI

The user interface of the .NET MAUI app is created using controls. These controls can be categorized as pages, layouts, and views.

A page is the top-level user interface element that usually occupies all the screens or windows. We introduced how to create pages using the Visual Studio template or `dotnet` command at the beginning of this chapter. Each page typically contains at least one layout element, which is used to organize the design of controls on a page.

Views are the UI objects to present, edit, or initiate commands in the user interface design. Please refer to the following Microsoft document about the controls in .NET MAUI: <https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/>.

We will introduce layouts in the next section. In this section, we'll go through some controls that will be frequently used in this book. Please refer to the preceding link for more details.

Label

`Label` is used to display single-line or multi-line text. It can display text with a certain format, such as color, space, text decorations, and even HTML text. To create a `Label`, we can use the simplest format, like so:

```
<Label Text="Hello world" />
```

Image

In the user interface design, we usually use icons to decorate other controls or display images as backgrounds. The `Image` control can display an image from a local file, a URI, an embedded resource, or a stream. The following code shows an example of how to create an `Image` control in the simplest form:

```
<Image Source="dotnet_bot.png" />
```

Editor

In our app, the users need to enter or edit a single line of text or multiple lines of text. We have two controls to serve this purpose: `Editor` and `Entry`.

`Editor` can be used to enter or edit multiple lines of text. The following is an example of the `Editor` control:

```
<Editor Placeholder="Enter your description here" />
```

Entry

`Entry` can be used to enter or edit a single line of text. To design a login page, we can use `Entry` controls to enter a username and password. When users interact with an `Entry`, the behavior of the keyboard can be customized through the `Keyboard` property. When users enter their passwords, the `IsPassword` property can be set to reflect the typical behavior on a login page. The following is an example of a password entry:

```
<Entry Placeholder="Enter your password" Keyboard="Text"
      IsPassword="True" />
```

ListView

In the user interface design, a common use case is to display a collection of data. In .NET MAUI, a few controls can be used to display a collection of data, such as `CollectionView`, `ListView`, and `CarouselView`. In our app, we will use `ListView` to display password entries, groups, and the content of an entry. We will introduce the usage of `ListView` when we introduce `ItemsPage`.

Layouts in .NET MAUI

To design user interface elements in a view or page, we usually use layout control as a container to define the presentation format. There are a few layouts that are core to .NET MAUI.

StackLayout

`StackLayout` organizes elements in a one-dimensional stack, either horizontally or vertically. It is often used as a parent layout, which contains other child layouts. The default orientation is vertical. However, we should not use `StackLayout` to generate a layout similar to a table by using nested `StackLayout` horizontally and vertically. The following code shows an example of bad practice:

```
<StackLayout>
  <StackLayout Orientation="Horizontal">
    <Label Text="Name:" />
    <Entry Placeholder="Enter your name" />
  </StackLayout>
  <StackLayout Orientation="Horizontal">
    <Label Text="Age:" />
    <Entry Placeholder="Enter your age" />
  </StackLayout>
</StackLayout>
```

```
    </StackLayout>
    <StackLayout Orientation="Horizontal">
        <Label Text="Address:" />
        <Entry Placeholder="Enter your address" />
    </StackLayout>
</StackLayout>
```

In the preceding code, we used a `StackLayout` as the parent layout, where the default orientation is vertical. Then, we nested multiple `StackLayout` controls with a horizontal orientation to generate a form to fill in. We should use the `Grid` control to do this.

`StackLayout` is a frequently used layout control. There are two sub-types of `StackLayout` that help us directly design the layout horizontally or vertically.

HorizontalStackLayout

`HorizontalStackLayout` is a one-dimensional horizontal stack. For example, we can generate a row like so:

```
<HorizontalStackLayout>
    <Label Text="Name:" />
    <Entry Placeholder="Enter your name" />
</HorizontalStackLayout>
```

VerticalStackLayout

`VerticalStackLayout` is a one-dimensional vertical stack. For example, we can display an error message after a form is submitted with an error like so:

```
<VerticalStackLayout>
    <Label Text="The Form Is Invalid" />
    <Button Text="OK"/>
</VerticalStackLayout>
```

Grid

`Grid` organizes elements in rows and columns. We can specify rows and columns with the `RowDefinitions` and `ColumnDefinitions` properties. In the previous example, we created a form where the user can enter their name, age, and address using a nested `StackLayout`. We can do this in the `Grid` layout like so:

```
<Grid>
    <Grid.RowDefinitions>
```

```
<RowDefinition Height="50" />
<RowDefinition Height="50" />
<RowDefinition Height="50" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="Auto" />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
<Label Text="Name: " />
<Entry Grid.Column="1"
      Placeholder="Enter your name" />
<Label Grid.Row="1" Text="Age: " />
<Entry Grid.Row="1" Grid.Column="1"
      Placeholder="Enter your age" />
<Label Grid.Row="2" Text="Address: " />
<Entry Grid.Row="2"
      Grid.Column="1"
      Placeholder="Enter your address" />
</Grid>
```

In the preceding example, we created a Grid layout with two columns and three rows.

FlexLayout

FlexLayout is similar to a StackLayout in that it displays child elements either horizontally or vertically in a stack. The difference is a FlexLayout can also wrap its children if there are too many to fit in a single row or column. As an example, we can create a FlexLayout with five labels in a row. If we specify the Direction property as Row, these labels will be displayed in one row. We can also specify the Wrap property, which can cause the items to wrap to the next row if there are too many items to fit in a row:

```
<FlexLayout Direction="Row" Wrap="Wrap">
  <Label Text="Item 1" Padding="10"/>
  <Label Text="Item 2" Padding="10"/>
  <Label Text="Item 3" Padding="10"/>
  <Label Text="Item 4" Padding="10"/>
  <Label Text="Item 5" Padding="10"/>
</FlexLayout>
```

AbsoluteLayout

`AbsoluteLayout` is a layout type that we can use to position elements using X, Y, width, and height.

The X and Y positions are relative to the top-left corner of the parent element. Width and height are concerned with the size of the child element.

In the following example, we are creating a `BoxView` control in the layout at (0, 0) with both width and height equal to 10:

```
<AbsoluteLayout Margin="20">
    <BoxView Color="Silver"
        AbsoluteLayout.LayoutBounds="0, 0, 10, 10" />
</AbsoluteLayout>
```

Navigation in the master-detail UI design

As shown in *Figure 3.4*, we are using a stacked master-detail pattern in our navigation. There is a flyout menu to display a list of pages. In the list of pages, a page of the `ItemsPage` type is used to display a list of password entries. When users click an entry, details about the password entry are shown in `ItemDetailPage`.

Let's review the implementation of `ItemsPage` and `ItemDetailPage`.

ItemDetailPage

In our app, `ItemDetailPage` is the detail page of the master-detail pattern, and it shows the content of an item. In `ItemDetailPage`, we simply present the `Item` data model. It looks very simple at the moment; we will enhance it gradually throughout this book:

Listing 3.3: `Item.cs` (<https://epa.ms/Item3-3>)

```
using System;

namespace PassXYZ.Vault.Models {
    public class Item {
        public string Id { get; set; }
        public string Text { get; set; }
        public string Description { get; set; }
    }
}
```

As shown in *Listing 3.3*, the `Item` class includes three properties called `ID`, `Text`, and `Description`. The instance of `Item` is loaded by the `LoadItemId()` function in `ItemDetailViewModel`, as shown here. We will discuss the MVVM pattern in the next chapter:

```
public async void LoadItemId(string itemId) {
    try {
        var item = await DataStore.GetItemAsync
            (itemId);
        Id = item.Id;
        Text = item.Text;
        Description = item.Description;
    }
    catch (Exception) {
        Debug.WriteLine("Failed to Load Item");
    }
}
```

Once the data has been loaded, we can present the data to the user in `ItemDetailPage.xaml`, as shown in *Listing 3.4*:

Listing 3.4: ItemDetailPage.xaml (<https://epa.ms/ItemDetailPage3-4>)

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com
    /dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com
        /winfx/2009/xaml"
    x:Class="PassXYZ.Vault.Views.ItemDetailPage"
    Title="{Binding Title}">

    <StackLayout Spacing="20" Padding="15">
        <Label Text="Text:" FontSize="Medium" />
        <Label Text="{Binding Text}" FontSize="Small"/>
        <Label Text="Description:" FontSize="Medium" />
        <Label Text="{Binding Description}"
            FontSize="Small"/>
    </StackLayout>
</ContentPage>
```

Listing 3.4 is the XAML file of `ItemDetailPage`. The content page of the item detail includes an instance of `StackLayout` and four instances of `Label`.

In `StackLayout`, the default orientation is `Vertical`, so the `Label` controls are organized vertically on the item detail page (see *Figure 3.4*). Both `Text` and `Description` are connected to the model data in the view model through data binding. We will introduce data binding in the next chapter.

ItemsPage

`ItemsPage` is the master page of the master-detail pattern in our app. It displays a list of items that we can explore.

Listing 3.5 is the implementation of `ItemsPage`. To display a list of items, a `ListView` control is used. `ListView` is a control used to display a scrollable vertical list of selectable data items:

Listing 3.5: ItemsPage.xaml (<https://epa.ms/ItemsPage3-5>)

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com
  /dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com
    /winfx/2009/xaml"
  x:Class="PassXYZ.Vault.Views.ItemsPage" ❶
  Title="{Binding Title}"
  xmlns:local="clr-namespace:
    PassXYZ.Vault.ViewModels" ❺
  xmlns:model="clr-namespace:PassXYZ.
    Vault.Models" ❻
  x:DataType="local:ItemsViewModel" ❷
  x:Name="BrowseItemsPage" ❸

  <ContentPage.ToolbarItems...>
  <StackLayout>
    <ListView x:Name="ItemsListView" ❹
      ItemsSource="{Binding Items}"
      VerticalOptions="FillAndExpand"
      HasUnevenRows="False"
      RowHeight="84"
```

```
RefreshCommand="{Binding LoadItems
    Command}"
IsPullToRefreshEnabled="true"
IsRefreshing="{Binding IsBusy,
    Mode=OneWay}"
CachingStrategy="RetainElement"
ItemSelected="OnItemSelected">
    <ListView.ItemTemplate>
        <DataTemplate...>
    </ListView.ItemTemplate>
</ListView>
</StackLayout>
</ContentPage>
```

Let's look at this code in more detail:

❶ **x:Class**: This is used to define the class name of a partial class between the markup and code-behind file. `PassXYZ.Vault.Views.ItemsPage` is the class name defined here.

❸ **x:Name**: While **x:Class** defines the class name in XAML, **x:Name** defines the instance name. We can refer to the `BrowseItemsPage` instance name in the code-behind file.

❷ **x:DataType**: When we set **x:DataType** to the appropriate type defined in the view model, compiled binding can be turned on, which can improve performance significantly. The view model that we refer to here is `ItemsViewModel`.

Besides the standard namespace, we defined two more namespaces so that we can refer to the objects in the view model ❹ and model ❺. We will discuss the view model and model in the next chapter.

❹ We define a `ListView` control to display the list of items. There are many properties in the `ListView` control. The following properties are the ones we have to define to use the `ListView` control:

- `ItemsSource`, of the `IEnumerable` type, specifies the collection of items to be displayed. It binds to `Items`, which is defined in the view model.
- `ItemTemplate`, of the `DataTemplate` type, specifies the template to apply to each item in the collection of items to be displayed.

In *Listing 3.5*, `DataTemplate` is collapsed. If we expand it, we will see the following code snippet. This is the default implementation from the Visual Studio template. The look and feel of this data template are not good enough, but we can improve it:

```
<DataTemplate>
    <ViewCell>
```

```

        <StackLayout Padding="10" x:DataType="model:Item">
            <Label Text="{Binding Text}"
                LineBreakMode="NoWrap"
                Style="{DynamicResource ListItemTextStyle}"
                FontSize="16" />
            <Label Text="{Binding Description}"
                LineBreakMode="NoWrap"
                Style="{DynamicResource
                    ListItemDetailTextStyle}"
                FontSize="13" />
        </StackLayout>
    </ViewCell>
</DataTemplate>

```

This `DataTemplate` implementation includes a `ViewCell` consisting of a `StackLayout` with two `Label` controls. We can see the preview in *Figure 3.4*.

The `DataTemplate` implementation must reference a `Cell` class to display items. There are built-in cells that can be used, as follows:

- `TextCell`, which displays primary and secondary text on separate lines.
- `ImageCell`, which displays an image with primary and secondary text on separate lines.
- `SwitchCell`, which displays text and a switch that can be switched on or off.
- `EntryCell`, which displays a label and text that's editable.
- `ViewCell`, which is a custom cell whose appearance is defined by a `View`. This cell type should be used when you want to fully define the appearance of each item in a `ListView`.

Typically, `SwitchCell` and `EntryCell` are only used in a `TableView` and won't be used in a `ListView`.

The preview of `ViewCell` in the preceding code snippet doesn't look very good. It is not easy to differentiate between `Text` and `Description`. In `KeePass`, we usually attach an icon to the password entry. We can enhance it using the new data template, like so:

```

<DataTemplate>
    <ViewCell>
        <Grid Padding="10" x:DataType="model:Item" >           ①
            <Grid.RowDefinitions>                                ②
                <RowDefinition Height="32" />

```

```

        <RowDefinition Height="32" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Grid Grid.RowSpan="2" Padding="10">                                     ③
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="32" />
        </Grid.ColumnDefinitions>
        <Image Grid.Column="0" Source="passxyz_logo.png"
            HorizontalOptions="Fill" VerticalOptions="Fill" />
    </Grid>
    <Label Text="{Binding Text}" Grid.Column="1"
        LineBreakMode="NoWrap" MaxLines="1"
        Style="{DynamicResource ListItemTextStyle}"
        FontAttributes="Bold" FontSize="Small" />
    <Label Text="{Binding Description}"
        Grid.Row="1" Grid.Column="1"
        LineBreakMode="TailTruncation" MaxLines="1"
        Style="{DynamicResource ListItemDetailTextStyle}"
        FontSize="Small" />
    </Grid>
</ViewCell>
</DataTemplate>

```

Let's look at this code in more detail:

- ① To make ViewCell look and feel better, we replaced the layout class from StackLayout to Grid. Grid is a layout that organizes its children into rows and columns.
- ② Since we want to display two rows with an icon at the left, we created a grid with two columns and two rows, as shown here:

	Text
	Description

Figure 3.5: Layout of an entry or a group

We can use different font styles for Text and Description so that users can easily differentiate them with visual effects.

- ③ To display the icon at the center of the first two columns, we merged the two rows into a `Grid` control. We can use the attached `Grid.RowSpan` property to merge rows.

A `Grid` can be used as a parent layout that contains other child layouts. To make the icon a specific size and at the center of the merged cell, we can use another `Grid` as the parent of the `Image` control. This child `Grid` contains only one row and one column with a specific size.

In the `Image` control, we can use a default image (`passxyz_logo.png`) from the resource. It can be customized after we introduce our model in the next chapter.

We can see the improved preview in *Figure 3.6*:

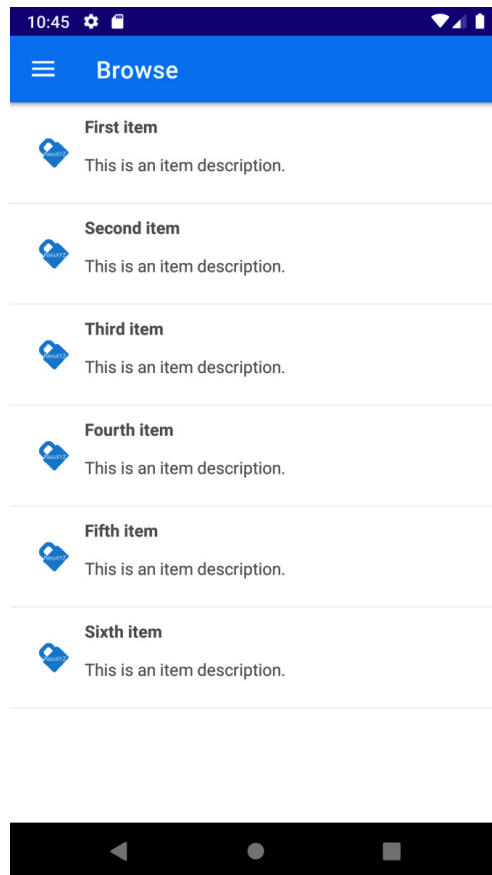


Figure 3.6: Improved ItemsPage

With that, we've learned the basics of user interface design using XAML. One common issue in the user interface design is supporting multiple languages. In the remainder of this chapter, we will learn how to support multiple languages in XAML user interface design.

Supporting multiple languages – localization

To support multiple languages, we can use the .NET built-in mechanism for localizing applications. In a XAML file, we can use the `x:Static` markup extension to use the string defined in resource files.

Creating a .resx file

Resource files are XML files with a `.resx` extension that are compiled into binary resource files during the build process. A resource file can be added by right-clicking the project node and selecting **Add > New Item... > Resources File**, as shown in *Figure 3.7*:

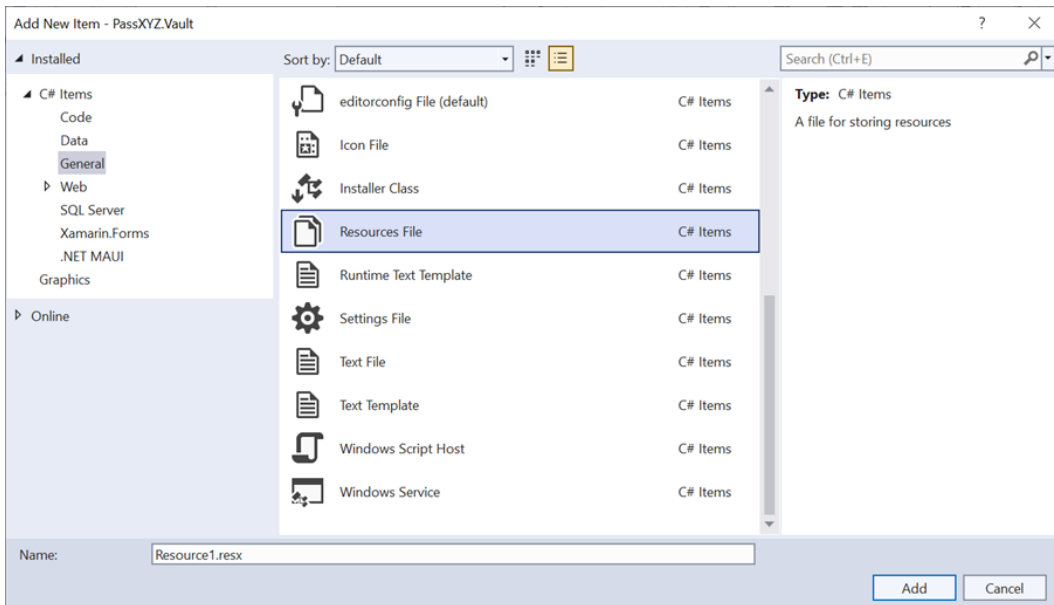


Figure 3.7: Creating a Resources File

We can create the `Resources.resx` resource file in the `Properties` folder.

To support different cultures, we can add additional resource files with cultural information as part of the resource file's name:

- `Resources.resx`: The resource file for the default culture, which we will set to `en-US` (US English) later
- `Resources.zh-Hans.resx`: The resource file for the `zh-Hans` culture, which is simplified Chinese
- `Resources.zh-Hant.resx`: The resource file for the `zh-Hant` culture, which is traditional Chinese

Once the resource file has been created, the following `ItemGroup` will be added to the project file:

```
<ItemGroup>
  <Compile Update="Properties\Resources.Designer.cs">
    <DesignTime>True</DesignTime>
    <AutoGen>True</AutoGen>
    <DependentUpon>Resources.resx</DependentUpon>
  </Compile>
</ItemGroup>
<ItemGroup>
  <EmbeddedResource Update="Properties\Resources.resx">
    <Generator>ResXFileCodeGenerator</Generator>
    <LastGenOutput>Resources.Designer.cs</LastGenOutput>
  </EmbeddedResource>
</ItemGroup>
```

To edit the resource file, we can click a resource file and edit it in the resource editor, as shown in *Figure 3.8*:

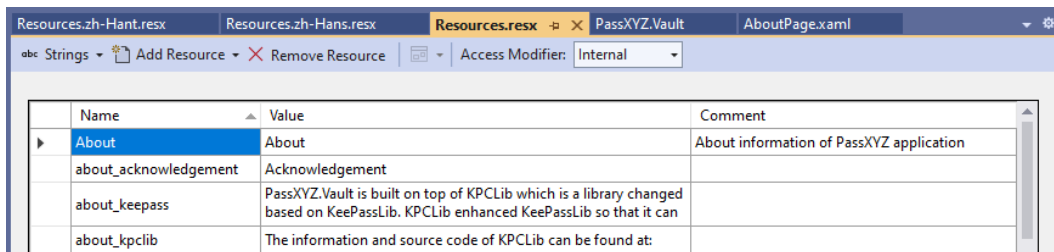


Figure 3.8: Resource editor

The resource file includes a list of key-value pairs for different languages:

- The **Name** field is the string name that we can refer to in either XAML or C# files
- The **Value** field contains the language-specific string that will be used according to the system language settings
- The **Comment** field is just used as a remark for the key-value pair

To specify the default language, we need to set the value of `NeutralLanguage` in `<PropertyGroup>` in the project file, as shown here:

```
<PropertyGroup>
  ...
```

```
<NeutralLanguage>en-US</NeutralLanguage>
...
</PropertyGroup>
```

In our project, we will use US English as the default culture, so `NeutralLanguage` is set to `en-US`.

Localizing text

Once we have set up the resource files, we can use localized content in our XAML file or C# files. We have five content pages in our project now. Let's modify `AboutPage` with localization support, as shown in *Listing 3.6*:

Listing 3.6: `AboutPage.xaml` (<https://epa.ms/AboutPage3-6>)

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com
  /dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="PassXYZ.Vault.Views.AboutPage"
  xmlns:res="clr-namespace:PassXYZ.Vault.Properties"
  Title="{Binding Title}">
  1

  <ContentPage.Resources...>
  <ScrollView>
    <StackLayout Margin="20">
      <Grid Padding="10"...>
        <StackLayout Padding="10" >
          <Label HorizontalOptions="Center"
            Text="{x:Static res:Resources.Appname}"
            FontAttributes="Bold" FontSize="22" />
            2
          <Label x:Name="AppVersion" HorizontalOptions
            ="Center"
            FontSize="Small" />
          <Grid HorizontalOptions="Center"...>
            <StackLayout...>
              </StackLayout>
            </StackLayout>
          </ScrollView>
        </ContentPage>
```

Text is localized using the generated Resources class. This class is named based on the default resource file name. In *Listing 3.6 AboutPage.xaml*, we added a new namespace ❶ for the Resources class:

```
xmlns:res ="clr-namespace:PassXYZ.Vault.Properties "
```

In the Label control ❷, to display our application name, we can refer to the resource string using the `x:Static` XAML markup extension, like so:

```
<Label HorizontalOptions="Center"
      Text="{x:Static res:Resources.Appname}"
      FontAttributes="Bold" FontSize="22" />
```

In *Listing 3.6*, we collapsed most of the source code to be concise. Please refer to the short URL of this book's GitHub repository to review the full source code.

We can use localized text in both XAML and C#. To use a resource string in C#, we can look at the Title property in *Listing 3.6*. The Title property of `AboutPage` is connected to the Title property in the `AboutViewModel` class. Let's see how we can use a resource string in *Listing 3.7*:

Listing 3.7: AboutViewModel.cs (<https://epa.ms/AboutViewModel3-7>)

```
using System;
using System.Windows.Input;
using Microsoft.Maui.Essentials;
using Microsoft.Maui.Controls;

using PassXYZ.Vault.Properties; ❶

namespace PassXYZ.Vault.ViewModels {
    public class AboutViewModel : BaseViewModel {
        public AboutViewModel() {
            Title = Properties.Resources.About; ❷
            OpenWebCommand = new Command(async () => await
                Browser.OpenAsync(Properties.Resources.about_url));
        }

        public ICommand OpenWebCommand { get; }
        public string GetStoreName()...
        public DateTime GetStoreModifiedTime()...
    }
}
```

As shown in *Listing 3.7*, ① we added the `PassXYZ.Vault.Properties` namespace first. ② We refer to the resource string as `Properties.Resources.About`.

After we update `AboutPage` with localization support, we can test it in the supported languages, as shown in *Figure 3.9*:

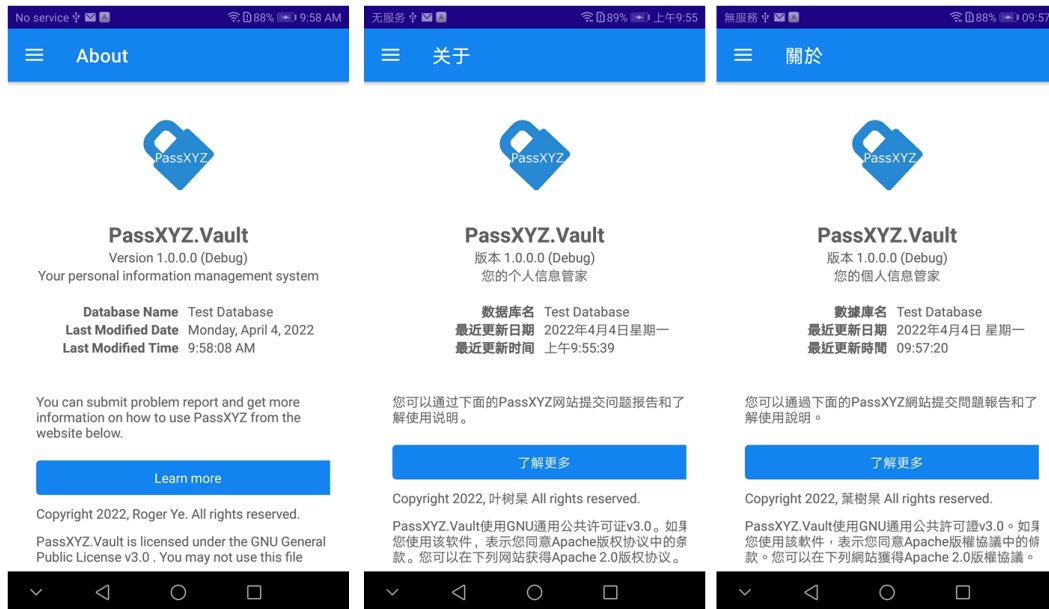


Figure 3.9: AboutPage in different languages

In `AboutPage`, many resource strings are used for localization. In *Listing 3.6* and *Listing 3.7*, we collapsed most of the code; you can refer to the short URL for this book's GitHub repository to review the source code online.

Summary

We learned about XAML syntax in this chapter. We used the knowledge we learned to improve the look and feel of `ItemsPage`. We will continue improving the user interface of other pages throughout this book. To support multiple languages, we learned how to support localization in .NET. We created .resx resource files for the US-en, zh-Hans, and zh-Hant cultures and used a XAML markup extension to enable multi-language support. Finally, we used `AboutPage` as an example to explain how to use localized text in both XAML and C#.

In the next chapter, we will continue improving our app by introducing MVVM and data binding.

Further reading

- **.NET Multi-platform App UI documentation**
- <https://learn.microsoft.com/en-us/dotnet/maui/>
- **XAML - .NET MAUI**
- <https://learn.microsoft.com/en-us/dotnet/maui/xaml/>
- **XAML markup extensions**
- <https://learn.microsoft.com/en-us/dotnet/maui/xaml/fundamentals/markup-extensions>
- **KeePass – An open source password manager**
- <https://keepass.info/>

4

Exploring MVVM and Data Binding

In the last chapter, we learned how to build **user interfaces (UIs)** using XAML. In this chapter, we will learn how to use the **Model-View-ViewModel (MVVM)** pattern and data binding in .NET MAUI app development. MVVM is a UI design pattern for decoupling UI and non-UI code. data binding is the key technology that MVVM relies on. We will improve the design of our app using MVVM and data binding. We will also replace the data model using open source libraries.

The following topics will be covered in this chapter:

- Understanding MVVM and MVC
- Data binding
- Improving the data model and service

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code of this chapter is available in the following branch at GitHub: <https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter04>.

The source code can be downloaded using the following Git command:

```
git clone -b chapter04 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development
```

Understanding MVVM and MVC

In software design, we usually follow and reuse good practices and design patterns. The **Model-View-Controller** (MVC) pattern is an approach to decoupling the responsibilities of a system. It can help to separate the implementation of the UI and the business logic into different parts.

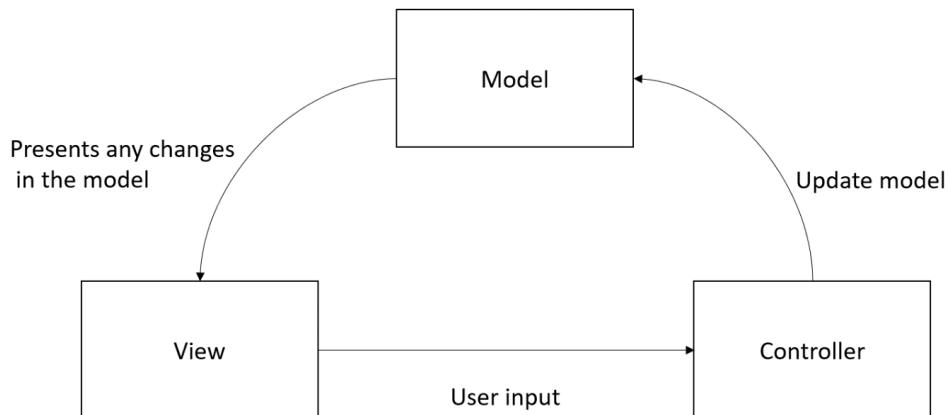


Figure 4.1: The MVC pattern

The MVC pattern, as shown in *Figure 4.1*, divides the responsibilities of the system into three distinct parts.

Model stores the application data and processes business logic. Model classes usually can be implemented as **Plain Old CLR Objects (POCOs)** or **Data Transfer Objects (DTOs)**. POCO is a class that doesn't depend on any framework-specific classes, so POCO classes can be used with LINQ or Entity Framework well. DTO is a subset of a POCO class that only contains data without logic or behavior. DTO classes can be used to pass data between layers. The model has no dependency on the view or the controller so it can be implemented and tested separately.

View presents the model information to the user and interacts with the user.

Controller updates the model and view in response to the user's action. Our understanding of the model and the view hasn't changed too much over time, but there have been different understandings and implementations of the controller since the MVC pattern was introduced.

Model-View-Presenter (MVP) is one of them. Later, Microsoft used MVVM and XAML in WPF, which is a variation of MVP. In Xamarin.Forms and .NET MAUI, XAML and the MVVM pattern are also used.

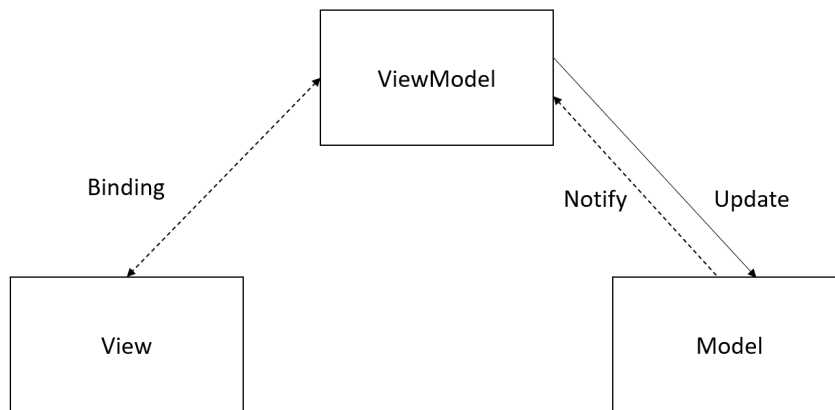


Figure 4.2: The MVVM pattern

As we can see in *Figure 4.2*, in MVVM, the view model is used to replace the controller. The differences between MVVM and MVC are as follows:

- Decoupling of view and model. The viewmodel is used to handle the communication between the view and the model. The view accesses the data and logic in the model via the viewmodel.
- Data binding between the view and viewmodel. Using data binding, changes to the view or viewmodel can automatically be updated in the other one. This can help to reduce the complexity of implementation.
- In both MVC and MVVM, the model can be tested separately. In MVVM, it is possible to design unit tests for the viewmodel as well.

When the view changes, the changes will be reflected in the viewmodel via data binding. The viewmodel will process the data changes in the model. Similarly, when the data changes in the model, the viewmodel is notified to update the view. The common solution for notifications is to install event handlers to notify the changes. With data binding, the implementation is simplified significantly.

MVVM in PassXYZ.Vault

In our app, `PassXYZ.Vault`, we use MVVM to handle the data exchange between the view and the viewmodel. As we can see in *Figure 4.3*, we have five XAML content pages and the same number of viewmodels defined. In our data model, we have an `Item` class, which is our model class, and it can be accessed through the `IDataStore` interface.

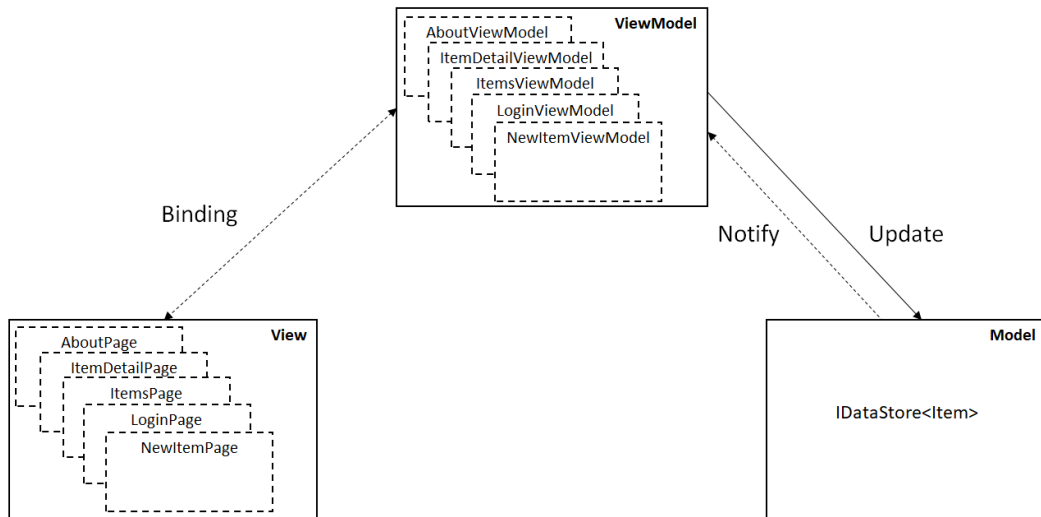


Figure 4.3: MVVM in PassXYZ.Vault

Data binding is used as the communication channel between views and viewmodels. The viewmodel will update the `Item` model via the `IDataStore` service interface. We will learn how to use data binding in the next section by analyzing the item detail page and viewmodel.

Data binding

Let's explore how MVVM and data binding works in our app. We can analyze `ItemDetailPage` and `ItemDetailViewModel` at the beginning of our journey. The following list includes the view, viewmodel, and model that we are going to explore:

- View – `ItemDetailPage`, see *Listing 3.4* in the previous chapter
- Viewmodel – `ItemDetailViewModel`, see *Listing 4.1*
- Model – `Item` (access through interface `IDataStore`), see *Listing 3.3* in the previous chapter

`ItemDetailPage` is a view used to display the content of an instance of `Item`. This instance is stored in the viewmodel. The UI elements presenting the content of `Item` are connected to the instance through data binding.

Data binding is used to link the properties of target and source objects. Here is a list of involved properties of target and source objects:

- **Target** – This is the UI element involved and this UI element has to be a child of `BindableObject`. The UI element used in `ItemDetailPage` is `Label`.

- **Target property** – This is the property of the target object. It is a `BindableProperty`. If the target is `Label`, as we mentioned here, the target property can be the `Text` property of `Label`.
- **Source** – This is the source object referenced by data binding. It is `ItemDetailViewModel` here.
- **Source object value path** – This is the path to the value in the source object. Here, the path is a viewmodel property, such as `Text` or `Description`.

Let's look at the following code in `ItemDetailPage`:

```
<StackLayout Spacing="20" Padding="15">
    <Label Text="Name:" FontSize="Medium" />
    <Label Text="{Binding Name}" FontSize="Small"/> ❶
    <Label Text="Description:" FontSize="Medium" />
    <Label Text="{Binding Description}"
        FontSize="Small"/> ❷
</StackLayout>
```

In the XAML here, there are two data binding source paths, which are `Name`, ❶, and `Description`, ❷. The binding target is `Label` and the target property is the `Text` property of `Label`. If we review the inheritance hierarchy of `Label`, it looks like so:

```
Object -> BindableObject -> Element -> NavigableElement -> VisualElement
-> View -> Label
```

We can see that `Element`, `VisualElement`, and `View` are the derivatives of `BindableObject`. The data binding target has to be a child of `BindableObject`.

The binding source is the `ItemDetailViewModel` viewmodel. `Name`, ❶, and `Description`, ❷, are properties of the viewmodel as shown in *Listing 4.1* here:

Listing 4.1: `ItemDetailViewModel.cs` (<https://epa.ms/ItemDetailViewModel4-1>)

```
using PassXYZ.Vault.Models;

namespace PassXYZ.Vault.ViewModels {
    [QueryProperty(nameof(ItemId), nameof(ItemId))]
    public class ItemDetailViewModel : BaseViewModel {
        private string itemId;
        private string name;
        private string description;
```

```

    public string Id { get; set; }

    public string Name {                                     ①
        get => name;
        set => SetProperty(ref name, value);
    }

    public string Description...                             ②
    public string ItemId...
    public async void LoadItemId(string itemId) {          ③
        try {
            var item = await DataStore.GetItemAsync
                (itemId);
            Id = item.Id;
            Name = item.Name;
            Description = item.Description;
        }
        catch (Exception) {
            Debug.WriteLine("Failed to Load Item");
        }
    }
}

```

The values of `Name`, ①, and `Description`, ②, are loaded from the model in the `LoadItemId()` method, ③. You may notice that the class is decorated by a `QueryPropertyAttribute` attribute. This is used to pass parameters during page navigation, and it will be introduced in the next chapter.

Let's use the following, *Table 4.1*, to summarize the data binding components in the code.

Data binding elements	Example
Target	Label
Target property	Text
Source object	ItemDetailViewModel
Source object value path	Name or Description

Table 4.1: Data binding settings

Having analyzed the preceding code, let us have a look at the syntax of the binding expression:

```
<object property="{Binding bindProp1=value1[,  
    bindPropN=valueN] *}" ... />
```

Binding properties can be set as a series of name-value pairs in the form of `bindProp=value`. For example, see the following:

```
<Label Text="{Binding Path=Description}" FontSize="Small"/>
```

The `Path` property is the default property, and it can be omitted if it is the first one in the property list as shown here:

```
<Label Text="{Binding Description}" FontSize="Small"/>
```

The `Source` property can be set to override `BindingContext`, which we will discuss shortly. There are many binding properties, and you can find the details by referring to Microsoft document about the `Binding` class here:

<https://learn.microsoft.com/en-us/dotnet/api/system.windows.data.binding?view=windowsdesktop-6.0>

When we set data binding to the target, we can use the following two members of the target class:

- The `BindingContext` property gives us the source object
- The `SetBinding` method specifies the target property and source property

In our case, we set the `BindingContext` property to an instance of `ItemDetailViewModel`, ❶, in the C# code-behind file of `ItemDetailPage` as shown in *Listing 4.2* here. It is set at the page level, and it applies to all binding targets for this page:

Listing 4.2: `ItemDetailPage.xaml.cs` (<https://epa.ms/ItemDetailPage4-2>)

```
using PassXYZ.Vault.ViewModels;  
using System.ComponentModel;  
using Microsoft.Maui;  
using Microsoft.Maui.Controls;  
  
namespace PassXYZ.Vault.Views  
{  
    public partial class ItemDetailPage : ContentPage  
    {
```

```
public ItemDetailPage()
{
    InitializeComponent();
    BindingContext = new ItemDetailViewModel(); ❶
}
void OnFieldSelected ...
}
```

Instead of using the Binding markup extension, we can also create the binding using the `SetBinding` method directly as done here:

```
<StackLayout Spacing="20" Padding="15">
    <Label Text="Text:" FontSize="Medium" />
    <Label x:Name="labelText" FontSize="Small"/> ❷
    <Label Text="Description:" FontSize="Medium" />
    <Label Text="{Binding Description}"
        FontSize="Small"/>
</StackLayout>
```

❷ In the XAML code, we removed the Binding markup extension and specified the instance name as `labelText`. In the C# code-behind file, we can call the `SetBinding()` method, ❸, in the constructor of `ItemDetailPage` to create the data binding for the `Text` property:

```
public ItemDetailPage()
{
    InitializeComponent();
    BindingContext = new ItemDetailViewModel();
    labelText.SetBinding(Label.TextProperty, "Text"); ❸
}
```

Binding mode

In this discussion, all the UI elements are `Label` objects, which are not editable for the user. This is one-way binding from the source to the target. In this kind of binding setup, we do not change target objects. The changes in the source object will cause updates in the target object.

There are four binding modes supported in .NET MAUI. Let's review them by referring to *Figure 4.4*.

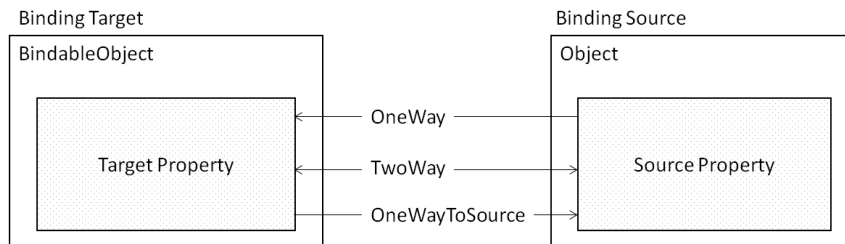


Figure 4.4: Binding mode

These binding modes are supported in .NET MAUI:

- **OneWay** binding is usually used in the case of presenting data to the user. In our app, we will retrieve a list of password entries and display this list on `ItemsPage`. When the user clicks an item in the list, the password details will show on `ItemDetailPage`. `OneWay` is used in both cases.
- **TwoWay** binding causes changes to either the source property or the target property to automatically update the other. In our app, when the user edits the fields of a password entry or when the user enters a username and password on `LoginPage`, the target UI Entry component and the source view model object are set with `TwoWay`.
- **OneWayToSource** is the reverse of the `OneWay` binding mode. When the target property is changed, the source property will be updated. When we add a new password entry on `NewItemPage`, we can use `OneWayToSource` instead of the `TwoWay` binding mode to improve performance.
- **OneTime** binding is a binding mode that is not shown in *Figure 4.4*. The target properties are initialized from the source properties, but any further changes to the source properties won't update the target properties. It is a simpler form of the `OneWay` binding mode with better performance.

If we don't specify the binding mode in data binding, the default binding mode is used. We can overwrite the default binding mode if it is needed.

In our `ItemsPage` code, we use the `ListView` control to display the list of password groups and entries, so we should set the `IsRefreshing` attribute to the `OneWay` binding mode:

```
IsRefreshing="{Binding IsBusy, Mode=OneWay}"
```

When we add a new item in `NewItemPage`, we use the `Entry` and `Editor` controls to edit the properties. We can use the `OneWayToSource` or `TwoWay` binding modes:

```
<Label Text="Text" FontSize="Medium" />
<Entry Text="{Binding Text, Mode=TwoWay}" FontSize="Medium"
/>
<Label Text="Description" FontSize="Medium" />
<Editor Text="{Binding Description, Mode=OneWayToSource}"
AutoSize="TextChanges" FontSize="Medium" Margin="0" />
```

Changing notifications in viewmodels

In *Figure 4.4*, we can see the data binding target is a derived class of `BindableObject`. Besides this requirement, in the data binding setup, both the data binding target and source also need to implement the `INotifyPropertyChanged` interface so that when the property changes, a `PropertyChanged` event is raised to notify the change.

In an MVVM pattern, the viewmodel is usually the data binding source and we need to implement the `INotifyPropertyChanged` interface in our viewmodels. If we do this for each viewmodel, there will be a lot of duplicated code. In a Visual Studio template, a `BaseViewModel` class, as we can see in *Listing 4.3*, is included in the boilerplate code and we use it in our app. Other viewmodels inherit this class:

Listing 4.3 BaseViewModel.cs (<https://epa.ms/BaseViewModel4-3>)

```
namespace PassXYZ.Vault.ViewModels;
public class BaseViewModel : INotifyPropertyChanged ❶
{
    public IDataStore<Item> DataStore =>
        DependencyService.Get<IDataStore<Item>>();

    bool isBusy = false;
    public bool IsBusy {
        get { return isBusy; }
        set { SetProperty(ref isBusy, value); } ❷
    }

    string title = string.Empty;
    public string Title {
```

```

    get { return title; }
    set { SetProperty(ref title, value); }
}

protected bool SetProperty<T>(ref T backingStore,
    T value,
    [CallerMemberName] string propertyName = "",
    Action onChanged = null) {
    if (EqualityComparer<T>.Default.Equals
        (backingStore, value))
        return false;

    backingStore = value;
    onChanged?.Invoke();
    OnPropertyChanged(propertyName);
    return true;
}

#region INotifyPropertyChanged
public event PropertyChangedEventHandler PropertyChanged;❹
protected void OnPropertyChanged([CallerMemberName]
    string propertyName = "") {❸
    var changed = PropertyChanged;
    if (changed == null)
        return;

    changed.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
}
#endregion
}

```

In the `BaseViewModel` class (*Listing 4.3*), we can see the following:

- ❶ `BaseViewModel` implements the `INotifyPropertyChanged` interface and this interface defines a single event, `PropertyChanged`, ❹.

- ❸ When a property is changed in the setter, the `OnPropertyChanged` method is called. In `OnPropertyChanged`, the `PropertyChanged` event is fired. A copy of the `PropertyChanged` event handler is stored in the changed local variable, so this implementation is safe in a multi-thread environment. When the `PropertyChanged` event is fired, it needs to pass the property name as a parameter to indicate which property is changed. The `CallerMemberName` attribute can be used to find the method name or property name of the caller, so we don't need to hardcode the property name.
- ❷ When we define a property in the viewmodel, the `OnPropertyChanged` method is called in the setter – but as you can see, in our code, we call `SetProperty<T>` instead of `OnPropertyChanged` directly. `SetProperty<T>` will do additional work before it calls `OnPropertyChanged`. It checks whether the value is changed. If there is no change, it will return and do nothing. If the value is changed, it will update the backing field and call `OnPropertyChanged` to fire the change event.

If we recall `ItemDetailViewModel` in *Listing 4.1*, it inherits from the `BaseViewModel` class. In the setter of the `Name` and `Description` properties, we call `SetProperty<T>` to set the values and fire the `PropertyChanged` event:

```
public string Name {  
    get => name;  
    set => SetProperty(ref name, value);  
}  
  
public string Description {  
    get => description;  
    set => SetProperty(ref description, value);  
}
```

In this section, we learned about data binding and the `INotifyPropertyChanged` interface. We need to create boilerplate code to define a property with change notification support. To simplify the code and autogenerate boilerplate code behind the scenes, we can use the MVVM Toolkit. Please find more information about the MVVM Toolkit in the *Further reading* section.

Having introduced some basic knowledge of XAML UI design, the MVVM pattern, and data binding, we can improve our app using the knowledge we just learned.

Improving the data model and service

To improve our app, let us review the use cases again. We are developing a cross-platform password manager app that is compatible with the popular **KeePass** database format. We have the following use cases:

- **Use case 1:** `LoginPage` – As a password manager user, I want to log in to the password manager app so that I can access my password data

- **Use case 2:** `AboutPage` – As a password manager user, I want to have an overview of my database and the app that I am using
- **Use case 3:** `ItemsPage` – As a password manager user, I want to see a list of groups and entries so that I can explore and examine my password data
- **Use case 4:** `ItemDetailPage` – As a password manager user, I want to see the details of a password entry after I select it in the list of password entries
- **Use case 5:** `NewItemPage` – As a password manager user, I want to add a password entry or create a new group in my database

These five use cases are inherited from the Visual Studio template, and they are sufficient for the user stories of our password manager app for the moment. We will improve our app using these user stories in this chapter.

We have explored the model, view, and viewmodel, but the model given here is too simple and is not sufficient for use in a password manager app:

```
public class Item
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
```

The major functionalities of our password manager app are encapsulated in the model layer. We will build our model using two .NET packages, `KPCLib` and `PassXYZLib`. These two packages include all the password management features we need.

KPCLib

The model that we will use is a library from KeePass called `KeePassLib`. Both `KeePass` and `KeePassLib` are built for .NET Framework, so they can only be used on Windows. I ported `KeePassLib` and rebuilt it as a .NET Standard 2.0 library packaged as `KPCLib`. `KPCLib` can be found at NuGet and GitHub here:

- NuGet: <https://www.nuget.org/packages/KPCLib/>
- GitHub: <https://github.com/passxyz/KPCLib>

`KPCLib` is used both as a package name and a namespace. The package of `KPCLib` includes two namespaces, `KeePassLib` and `KPCLib`. The `KeePassLib` namespace is the original one from KeePass with the following changes:

- Updated and built for .NET Standard 2.0
- Updated `PwEntry` and `PwGroup` to be classes derived from the `Item` abstract class

In the `KPCLib` namespace, an `Item` abstract class is defined. The reason I created a new class and made it the parent class of `PwEntry` and `PwGroup` is due to the navigation design difference between KeePass and PassXYZ.Vault.

If we look at the UI of KeePass in *Figure 4.5*, we can see that it is a classic Windows desktop UI. The navigation is designed around a tree view like Windows Explorer.

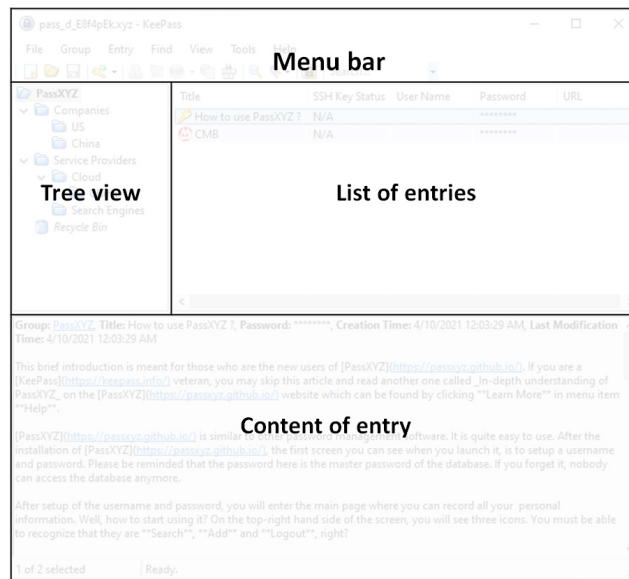


Figure 4.5: KeePass UI

Two classes, `PwGroup` and `PwEntry`, behave like directories and files. A `PwGroup` instance is just like a directory, and it includes a list of children – `PwGroup` and `PwEntry`. All `PwGroup` instances display in a tree view on the right-hand panel. When a `PwGroup` instance is selected, the list of `PwEntry` in this group is shown on the right-hand panel. `PwEntry` includes the content of a password entry, such as a username and password. The content of `PwEntry` is displayed on the bottom panel.

In the `PassXYZ.Vault` UI design, we use a .NET MAUI Shell template. It is an implementation of a stacked Master-Detail pattern. In the stacked Master-Detail pattern, a single list is used to display items. In this case, the instances of both `PwGroup` and `PwEntry` can be displayed in the same list. After an item is selected, we will take an action according to the type of the item.

Abstraction of PwGroup and PwEntry

To work with the PassXYZ.Vault UI design better, we can abstract PwGroup and PwEntry as Item abstract class, as shown in *Figure 4.6*.

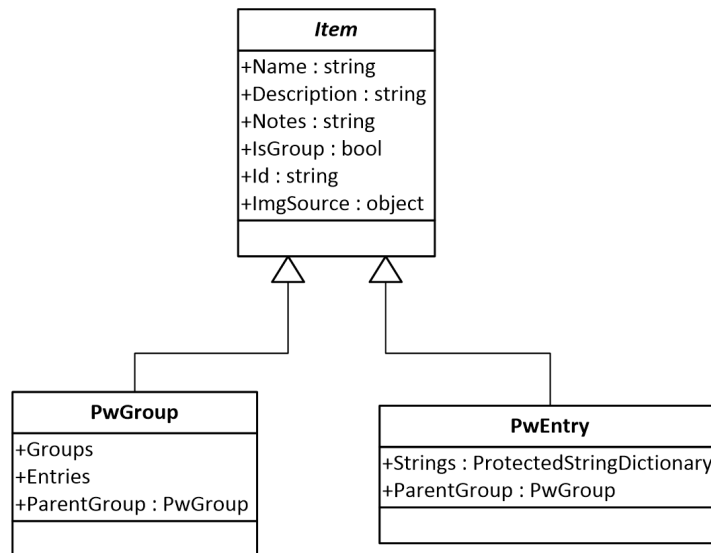


Figure 4.6: Class diagram of Item

Referring to this UML class diagram in *Figure 4.6* and the source code of `Item.cs` in *Listing 4.4*, we can see the following properties are defined in the `Item` abstract class. These properties are implemented in both `PwEntry` and `PwGroup`:

- ① `Name` – the `Item` name
- ② `Description` – the `Item` description
- ③ `Notes` – `Item` comments defined by the user
- ④ `IsGroup` – `true` if the instance is `PwGroup` or `false` if it is `PwEntry`
- ⑤ `Id` – ID of the instance (a unique value that is like the primary key in a database)
- ⑥ `ImgSource` – image source of the icon (both `PwGroup` and `PwEntry` can have an associated icon)
- ⑦ `LastModificationTime` – the last modification time of the item

- ⑧ Item implements the `INotifyPropertyChanged` interface, and it can work well in the MVVM model for data binding:

Listing 4.4: Item.cs (<https://epa.ms/Item4-4>)

```
using System.Text;

namespace KPCLib
{
    public abstract class Item : INotifyPropertyChanged ⑧
    {
        public abstract DateTime LastModificationTime {get;
            set;};} ⑦
        public abstract string Name { get; set; } ①
        public abstract string Description { get; } ②
        public abstract string Notes { get; set; } ③
        public abstract bool IsGroup { get; } ④
        public abstract string Id { get; } ⑤
        virtual public Object ImgSource { get; set; } ⑥

        #region INotifyPropertyChanged ...
        }
    }
}
```

PassXYZLib

To use `KeePassLib` in `PassXYZ.Vault`, we need to use some .NET MAUI APIs to extend the functionalities required of our app. To separate the business logic from the UI and extend the functionalities of `KeePassLib` for .NET MAUI, a .NET MAUI class library, `PassXYZLib`, is created to encapsulate the extended model in a separate library. `PassXYZLib` is both a package name and a namespace.

To add `PassXYZLib` to our project, we can add it to a `PassXYZ.Vault.csproj` project file, as seen here:

```
<ItemGroup>
  <PackageReference Include="PassXYZLib"
    Version="2.0.2" />
</ItemGroup>
```

We can also add a PassXYZLib package from the command line here. From the command line, go to the project folder and execute this command to add the package:

```
dotnet add package PassXYZLib
```

Updating the model

After we add a PassXYZLib package to the project, we can access the KPCLib, KeePassLib, and PassXYZLib namespaces. To replace the current model, we need to remove the Models/Item.cs file from the project.

After that, we need to replace the PassXYZ.Vault.Models namespace with KPCLib.

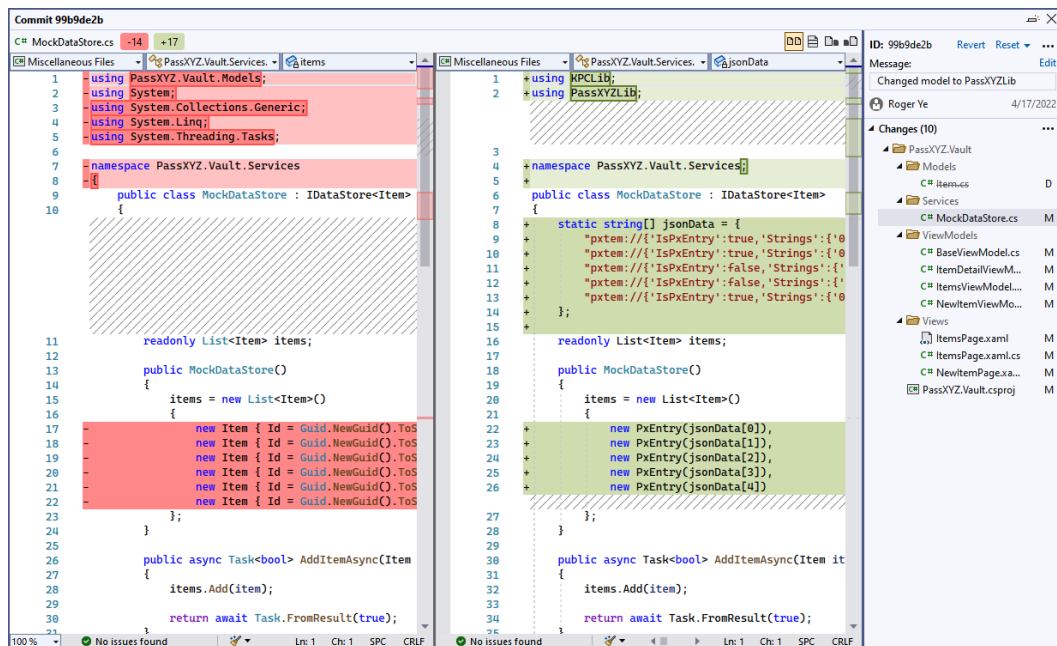


Figure 4.7: Updating the model from PassXYZ.Vault.Models to KPCLib (<https://bit.ly/3uXV17H>)

In the commit history, *Figure 4.7*, we can see that there are four view models, and three views are changed. All changes are namespace changes so we don't need to explain more about them.

Updating the service

The major changes can be found in `MockDataStore.cs`. In the `MockDataStore` class, we changed the namespace and the mock data initialization.

To decouple the model from the rest of the system, we use an `IDataStore` interface to encapsulate the actual implementation. At this stage, we use mock data to implement the service for testing, so the `MockDataStore` class is used. We will replace it with the actual implementation in *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*, using dependency injection.

Dependency inversion and dependency injection

We will learn about the **Dependency Inversion Principle (DIP)**, which is one of the SOLID design principles, in *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*. We will learn how to use dependency injection to manage the mapping of `IDataStore` interface to the actual implementation.

In the original code, we created new instances of `PassXYZ.Vault.Models.Item` to initialize mock data. After we replace the model, we cannot create `KPCLib.Item` directly, since it is an abstract class. Instead, we can create new instances of `PxEntry` using JSON data and assign `PxEntry` instances to the `Item` list:

```
Static string[] jsonData =...;
readonly List<Item> items;
public MockDataStore() {
    items = new List<Item>() {
        new PxEntry(jsonData[0]),
        new PxEntry(jsonData[1]),
        new PxEntry(jsonData[2]),
        new PxEntry(jsonData[3]),
        new PxEntry(jsonData[4])
    };
}
```

To create the instances of an abstract class, the factory pattern can be used. To make the testing code simple, we did not use it here. The factory pattern is used in the actual implementation later in this book.

We have replaced the model in the sample code with our own model now. With this change, we can improve `ItemsPage` and `ItemDetailPage` to reflect the updated model.

We will update the view and viewmodel using data binding to collections in the next section.

Binding to collections

In the previous section, we introduced some basic knowledge of data binding, and we also replaced the model using `PassXYZLib`. When we introduced data binding, we used `ItemDetailPage` and `ItemDetailViewModel` to explain how to bind the source property to the target property. For

the item detail page, we created data binding from one source to one target. However, there are many cases in which we need to bind a data collection to the UI, such as `ListView` or `CollectionView`, to display a group of data.

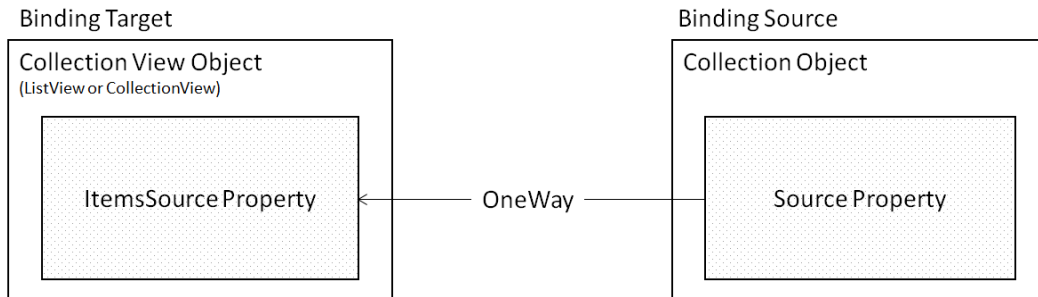


Figure 4.8: Binding to collections

As we can see in *Figure 4.8*, when we create a data binding from a collection object to a collection view, the `ItemsSource` property is the one to use. In .NET MAUI, collection views such as `ListView` and `CollectionView` can be used and both have an `ItemsSource` property.

For the collection object, we can use any collection that implements the `IEnumerable` interface. However, the changes to the collection object may not be able to update the UI automatically. In order to update UI automatically, the source object needs to implement the `INotifyCollectionChanged` interface.

We can implement our collection object with the `INotifyCollectionChanged` interface, but the simplest approach is to use the `ObservableCollection<T>` class. If any item in the observable collection is changed, the bound UI view is notified automatically.

With this in mind, let's review the class diagram of our models, viewmodels, and views as shown in *Figure 4.9*:

- **Model:** `Item`, `PwEntry`, `PwGroup`, `Field`
- **View Model:** `ItemsViewModel`, `ItemDetailViewModel`
- **View:** `ItemsPage`, `ItemDetailPage`

When we display a list of items to the user, the user may take action on the selected item. If the item is a group, we will show the groups and entries in an instance of `ItemsPage`. If the item is an entry, we will show the content of the entry on a content page, which is an instance of `ItemDetailPage`. On `ItemDetailPage`, we display a list of fields to the user. Each field is a key value pair and is implemented as an instance of `Field` class.

In summary, we display two kinds of lists to the user – a list of items or a list of fields. The list of items is shown in `ItemsPage` and the list of fields is shown in `ItemDetailPage`.

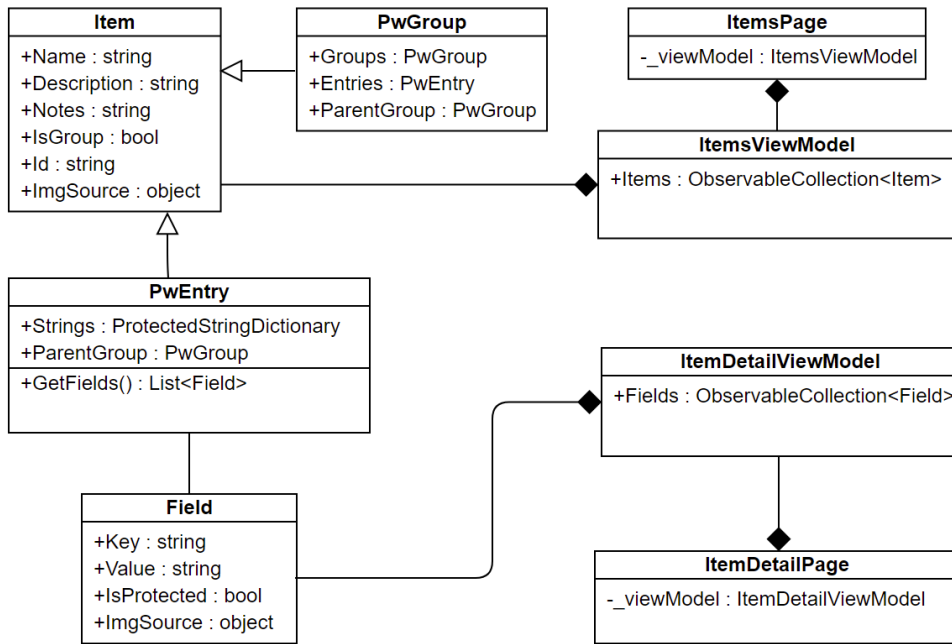


Figure 4.9: Class diagram of the model, view, and viewmodel

In this class diagram, we can see both `PwEntry` and `PwGroup` are derived from `Item`. There is a list of items in `ItemsViewModel` and there is a list of fields in `ItemDetailViewModel`. In the views, `ItemsPage` contains a reference to `ItemsViewModel`, and `ItemDetailPage` contains a reference to `ItemDetailViewModel`.

After we refine our design, we can look at the implementation. We will review the implementation of `ItemDetailViewModel` and `ItemDetailPage` to verify the design change:

```

[QueryProperty(nameof(ItemId), nameof(ItemId))]
public class ItemDetailViewModel : BaseViewModel {
    private string itemId;
    private string description;
    public string Id { get; set; }
    public ObservableCollection<Field> Fields { get; set; } ❶

    public string Description ...
    public string ItemId ...
}
  
```

```
public ItemDetailViewModel() {  
    Fields = new ObservableCollection<Field>();  
}  
public async void LoadItemId(string itemId) {  
    try {  
        var item = await DataStore.GetItemAsync(itemId);  
        Id = item.Id;  
        Title = item.Name;  
        Description = item.Description;  
  
        if (!item.IsGroup) {  
            PwEntry dataEntry = (PwEntry)item;  
            Fields.Clear();  
            List<Field> fields = dataEntry.GetFields(GetImage:  
                FieldIcons.GetImage);  
            foreach (Field field in fields) {  
                Fields.Add(field);  
            }  
        }  
    }  
    catch (Exception) {  
        Debug.WriteLine("Failed to Load I"em");  
    }  
}
```

As shown in the code here, we can see the difference in `ItemDetailViewModel` compared to *Listing 4.1* at the beginning of this chapter:

- ❶ A `Fields` property is defined as the `ObservableCollection<Field>` type to hold the `Field` list
- ❷ The `Fields` variable is initialized in the constructor of `ItemDetailViewModel`
- ❸ The `item` type of variable is `PwEntry` here and we can cast it to a `PwEntry` instance
- ❹ We can get the list of fields by calling an extension method, `GetFields()`, which is defined in the `PassXYZLib` library

Having reviewed the changes in `ItemDetailViewModel`, let's review the changes in `ItemDetailPage` in *Listing 4.5*:

Listing 4.5: `ItemDetailPage.xaml` (<https://epa.ms/ItemDetailPage4-5>)

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com
  /dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/
    winfx/2009/xaml"
  x:Class="PassXYZ.Vault.Views.ItemDetailPage"
  xmlns:local="clr-namespace:PassXYZ.
    Vault.ViewModels"
  xmlns:model="clr-namespace:KPCLib;
    assembly=KPCLib" ①
  x:DataType="local:ItemDetailViewModel"
  Title="{Binding Title}">

  <StackLayout>
    <ListView x:Name="FieldsListView"
      ItemsSource="{Binding Fields}" ②
      VerticalOptions="FillAndExpand"
      HasUnevenRows="False"
      RowHeight="84"
      IsPullToRefreshEnabled="true"
      IsRefreshing="{Binding IsBusy, Mode=
        OneWay}"
      CachingStrategy="RetainElement"
      ItemSelected="OnFieldSelected">

      <ListView.ItemTemplate...> ③
      <ListView.Footer>
        <StackLayout Padding="5" Orientation=
          "Horizontal">
          <Label Text="{Binding Description}
            .../>
        </StackLayout>
```

```

        </ListView.Footer>
    </ListView>
</StackLayout>

</ContentPage>

```

In `ItemDetailPage`, we can see there are many changes compared to *Listing 3.4* in *Chapter 3, User Interface Design with XAML*. `ListView` is used to display the fields in an entry:

- ① To use `Field` in `DataTemplate`, a `xmlns:model` namespace is added. Since the `Field` class is in a different assembly, we need to specify the assembly's name as follows:

```

xmlns:model="clr-
namespace:KPCLib;assembly=KPCLib"

```

- ② We bind the `Fields` property to the `ItemsSource` property of `ListView`.
- ③ `DataTemplate` is used to define the appearance of each item in `ListView`. It is collapsed in *Listing 4.5*.

Let's expand it and review the implementation of `DataTemplate` in this code block:

```

<DataTemplate>
    <ViewCell>
        <Grid Padding="10" x:DataType="model:Field" >           ❶
            <Grid.RowDefinitions...>
            <Grid.ColumnDefinitions...>
            <Grid Grid.RowSpan="2" Padding="10">
                <Grid.ColumnDefinitions...>
                <Image Grid.Column="0" Source="{Binding ImgSource}"
                    HorizontalOptions="Fill"
                    VerticalOptions="Fill" />           ❷
            </Grid>
            <Label Text="{Binding Key}" Grid.Column="1".../>       ❸
            <Label Text="{Binding Value}" Grid.Row="1"
                Grid.Column="1".../>                               ❹
        </Grid>
    </ViewCell>
</DataTemplate>

```

In `DataTemplate`, the layout of each field is defined in a `ViewCell` element. In the `ViewCell` element, we defined a `2x2 Grid` layout. The first column is used to display the field icon. The key and value in the field are displayed in the second column with two rows:

- **❶** The `x:DataType` attribute in the `Grid` layout is set to `Field` and the following data binding in `Grid` will refer to the property of `Field`. The `Field` class is defined in our model, which is in the `KPCLib` package.
- **❷** To display the field icon, the `Source` property of the `Image` control is set to the `ImgSource` property of `Field`.
- **❸,❹** Both the `Key` property and the `Value` property of `Field` are assigned to the `Text` property of the `Label` control.

With this analysis, we learned how to create data binding for a collection. The data binding used in `ItemsPage` and `ItemsViewModel` is similar to this implementation. The difference is we use a collection of `Field` here and a collection of `Item` classes is used in `ItemsPage`. Having completed the changes, we can see the improvement of the UI in *Figure 4.10*.

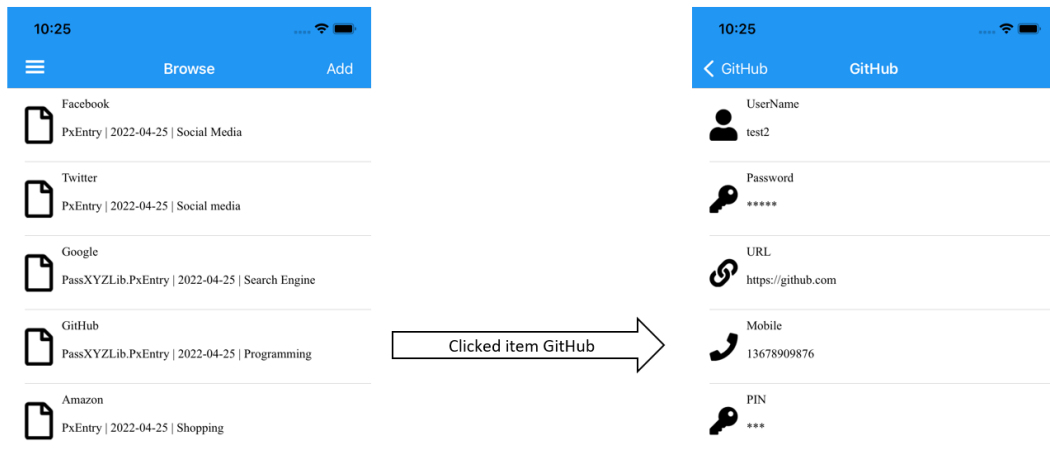


Figure 4.10: Improved `ItemsPage` and `ItemDetailPage`

In the improved UI, we display a list of items on `ItemsPage` (on the left). The items in the list can be entries (such as on Facebook, Twitter, or Amazon), or groups, which we will see in the next chapter.

When the user clicks on an item, such as **GitHub**, we will display it on `ItemDetailPage` (on the right). On the item detail page, the information about this account (**GitHub**) is shown.

Having introduced the new data model, the design hasn't changed much. We improved the UI to make it more meaningful, but most of the complexity is hidden in our model libraries – `KPCLib` and `PassXYZLib`. This is the benefit that we can see by using the MVVM pattern to separate the model (business logic) from the UI design.

Summary

In this chapter, we learned about the MVVM pattern and applied it to our app development. One key feature of the MVVM pattern is data binding between the view and viewmodel. We learned about data binding and used it in the implementation of our app.

We also improved the model in this chapter. We improved it by introducing two packages – `KPCLib` and `PassXYZLib`. We replaced the model in the sample code with the model in these two packages. We updated the UIs of `ItemsPage` and `ItemDetailPage` to reflect the changes to the model.

In the next chapter, we will refine our user stories and continue improving the UI using our knowledge of Shell and navigation.

Further reading

- *Introduction to the MVVM Toolkit*: <https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/>
- KeePass is a free open source password manager: <https://keepass.info/>

5

Navigation using .NET MAUI Shell and NavigationPage

In the previous chapter, we introduced the MVVM pattern and data binding. We improved the user interface design and introduced our data model. In our app, we can select a page from the flyout menu, and we can switch to the item detail when an item is selected. This is part of the navigation mechanism in .NET MAUI. In this chapter, we will dive deeper into the navigation design, and we will learn how navigation works in .NET MAUI.

The following topics will be covered in this chapter:

- Implementing navigation
- Using Shell
- Improving design and navigation

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code for this chapter is available in the following branch on GitHub: <https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter05>.

The source code can be downloaded using the following `git` command:

```
git clone -b chapter05 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development
```

Implementing navigation

In this chapter, we are going to implement the navigation logic of our password manager app. It will include the following functionalities:

- Logging in and connecting to the database
- Exploring data in the password database

Navigation design has a significant impact on the user experience. In .NET MAUI, there is a built-in mechanism to help developers implement the navigation efficiently. As we saw in the previous chapters, we can use `Shell` in our app. In this chapter, we will learn about `Shell` and improve our app with features provided by `Shell`. Before we dive into `Shell`, we will learn the basic navigation mechanism in .NET MAUI.

There are two most common ways to implement navigation – hierarchical and modal:

- **Hierarchical navigation** provides a navigation experience where the user can navigate through pages, both forward and backward. This pattern typically uses a toolbar or navigation bar at the top of the screen to display an Up or Back button in the top-left corner. It usually maintains a LIFO stack of pages to handle the navigation. **LIFO** stands for **last in, first out**, which means the last page to enter is the first one to pop out.
- **Modal navigation** is different from hierarchical navigation in terms of how users can respond to it. If a modal page is displayed on the screen, the users must complete or cancel the required task on the page before they can take other actions. The users cannot navigate away from modal pages before the required task is completed or canceled.

INavigation interface and NavigationPage

In .NET MAUI, both hierarchical navigation and model navigation are supported through the `INavigation` interface. The `INavigation` interface is supported by a special page called `NavigationPage`. `NavigationPage` is used to manage the navigation of a stack of other pages. The inheritance hierarchy of `NavigationPage` looks like this:

```
Object > BindableObject > Element > NavigableElement > VisualElement >
Page > NavigationPage
```

`NavigableElement` defines a property called `Navigation` that implements the `INavigation` interface. This inherited property can be called from any `VisualElement` or `Page` for navigation purposes, as shown here:

```
public Microsoft.Maui.Controls.INavigation Navigation { get; }
```

To use `NavigationPage`, we must add the first page to a navigation stack as the root page of the application. We can see an example of this in the following code snippet:

```
public partial class App : Application
{
    ...
    public App ()
    {
        InitializeComponent();
        MainPage = new NavigationPage (new TheFirstPage());
    }
    ...
}
```

We build the navigation stack in the constructor of the `App` class, which is a derived class of `Application`. `TheFirstPage`, which is a derived class of `ContentPage`, is pushed onto the navigation stack.

Using the navigation stack

There are two ways to navigate to or from a page. When we want to browse a new page, we can add the new page to the navigation stack. This action is called a **push**. If we want to go back to the previous page, we can **pop** the previous page from the stack:

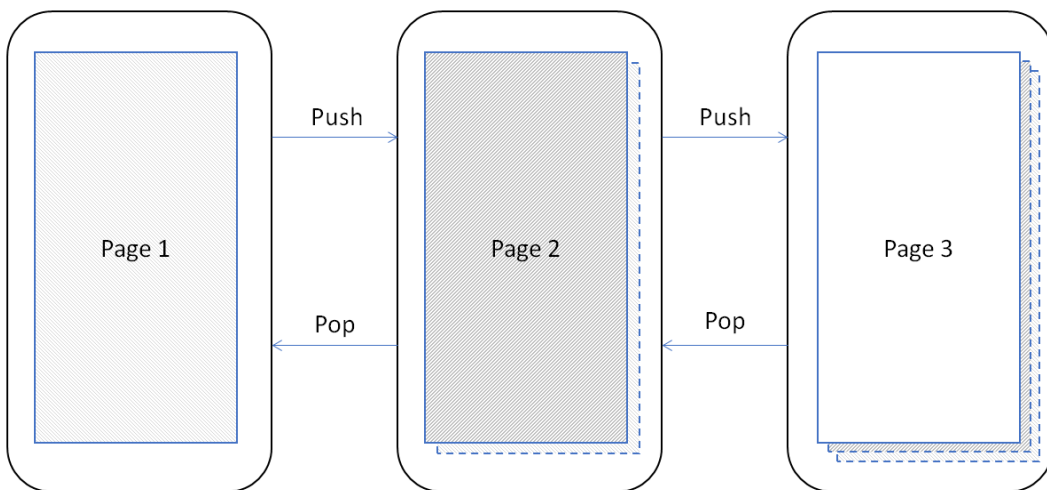


Figure 5.1: Push and pop

As shown in *Figure 5.1*, we can use the `PushAsync()` or `PopAsync()` method in the `INavigation` interface to change to a new page or go back to the previous page, respectively.

If we are on `Page1`, we can change to `Page2` with the `GotoPage2()` event handler. In this function, we are pushing the new page, `Page2`, to the stack:

```
async void GotoPage2 (object sender, EventArgs e) {  
    await Navigation.PushAsync(new Page2());  
}
```

Once we are on `Page2`, we can go back with the `BackToPage1()` event handler. In this function, we are popping the previous page from the stack:

```
async void BackToPage1 (object sender, EventArgs e) {  
    await Navigation.PopAsync();  
}
```

In the preceding example, we navigated to a new page using the hierarchical navigation method. To display a modal page, we can use the modal stack. For example, in our app, if we want to create a new item in `ItemsPage`, we can call `PushModalAsync()` in `ItemsViewModel`:

```
await Shell.Current.Navigation.  
PushModalAsync(NewItemPage(type));
```

After the new item has been created, we can call `PopModalAsync()` in `NewItemViewModel`:

```
_ = await Shell.Current.Navigation.PopModalAsync();
```

On the modal page's `NewItemPage`, we cannot navigate to other pages before we complete or cancel the task. Both `PopAsync()` and `PopModalAsync()` return an awaitable task of the `Task<Page>` type.

Manipulating the navigation stack

In hierarchical navigation, we can not only push or pop pages from the stack, but we can also manipulate the navigation stack.

Inserting a page

We can insert a page into the stack using the `InsertPageBefore()` method:

```
public void InsertPageBefore (Page page, Page before);
```

The following are two parameters of `InsertPageBefore()`:

- `page`: This is the page to be added
- `before`: This is the page before which the page is inserted

In *Figure 5.1*, when we are at `Page2`, we can insert another page, `Page1`, before it:

```
Navigation.InsertPageBefore(new Page1(), this);
```

Removing a page

We can also remove a specific page from the stack using the `RemovePage()` method:

```
public void RemovePage (Page page);
```

In *Figure 5.1*, given we have a reference of `Page2` when we are at `Page3`, we can remove `Page2` from the stack. After `PopAsync()` is called, we will be back at `Page1`:

```
// the reference page2 is an instance of Page2
Navigation.RemovePage(page2);
await Navigation.PopAsync();
```

With that, we have learned how to build a navigation stack using `NavigationPage`. Once we have a navigation stack, we can use the `INavigation` interface to perform navigation actions. For a simple application, this may be good enough. However, there will be a lot of work involved for a complex application. We have a better choice in .NET MAUI known as `Shell`. With `Shell`, we can provide a better navigation experience to the users with less work.

Using Shell

The `INavigation` interface and `NavigationPage` provide basic navigation functionalities. If we rely on them only, we have to create complicated navigation mechanisms by ourselves. In .NET MAUI, there are built-in page templates to choose from, and they can provide different navigation experiences.

As shown in the class diagram in *Figure 5.2*, there are built-in pages available for different use cases. All these pages – `TabbedPage`, `ContentPage`, `FlyoutPage`, `NavigationPage`, and `Shell` – are derived classes of `Page`:

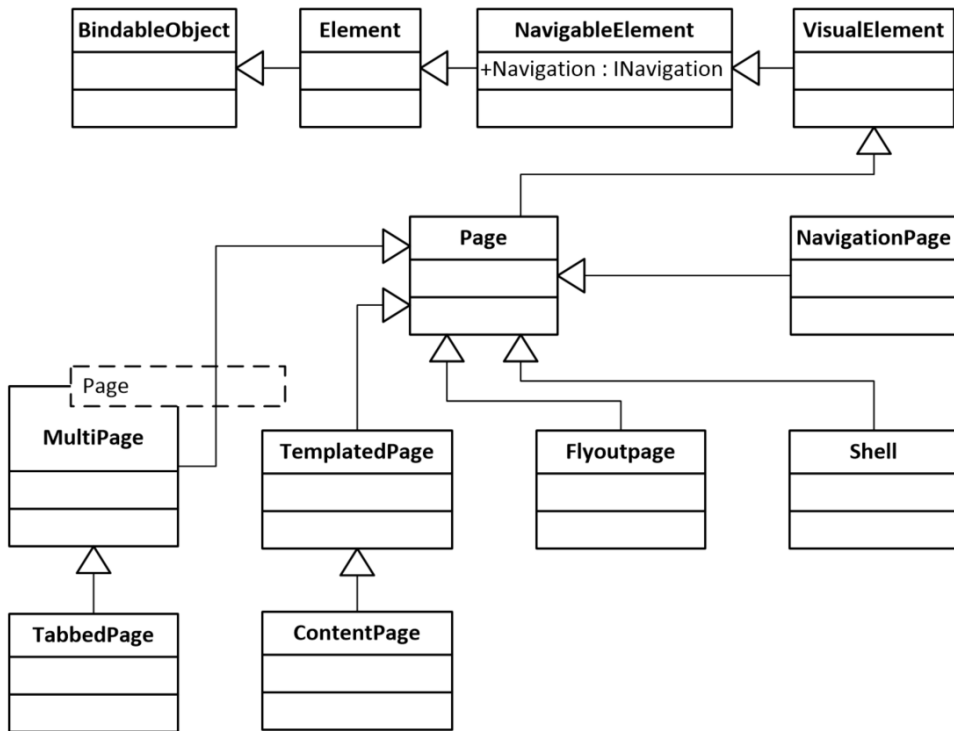


Figure 5.2: Class diagram of the built-in pages in .NET MAUI

ContentPage, **TabbedPage**, and **FlyoutPage** can be used to create various user interfaces per your requirements.

- **ContentPage** is the most used page and can include any layout and view elements. It is suitable in the case of a single-page design.
- **TabbedPage** can be used to host multiple pages. Each child page can be selected by a series of tabs at the top or bottom of the page.
- **Flyoutpage** can display a list of items, which is similar to the menu items in a desktop application. The user can navigate to individual pages through the items in the menu.

Even though **Shell** is also a derived class of **Page**, it includes a common navigation user experience, which can make developers' lives easier. It helps the developers by reducing the complexity of application development with highly customizable and rich features in one place.

Shell provides the following features:

- A single place to describe the visual hierarchy of an app
- A highly customizable common navigation user experience

- A URI-based navigation scheme that is very similar to what we have in a web browser
- An integrated search handler

The top-level building blocks of `Shell` are flyouts and tabs. We can use flyouts and tabs to create the navigation structure of our app.

Flyout

A flyout can be used as the top-level menu of a `Shell` app. In our app, we must use both flyouts and tabs to create the top-level navigation design. We will explore flyouts in this section; in the next section, we will discuss how to use tabs in our app.

In *Figure 5.3*, we can see what the flyout looks like in our app. From the flyout menu, we can switch to `AboutPage`, `ItemsPage`, or `LoginPage`. To access the flyout menu, we can either swipe from the left of the screen or click the flyout icon, which is the hamburger icon ①. When we click **Root Group** ② in the flyout menu, we will see a list of password entries or groups:

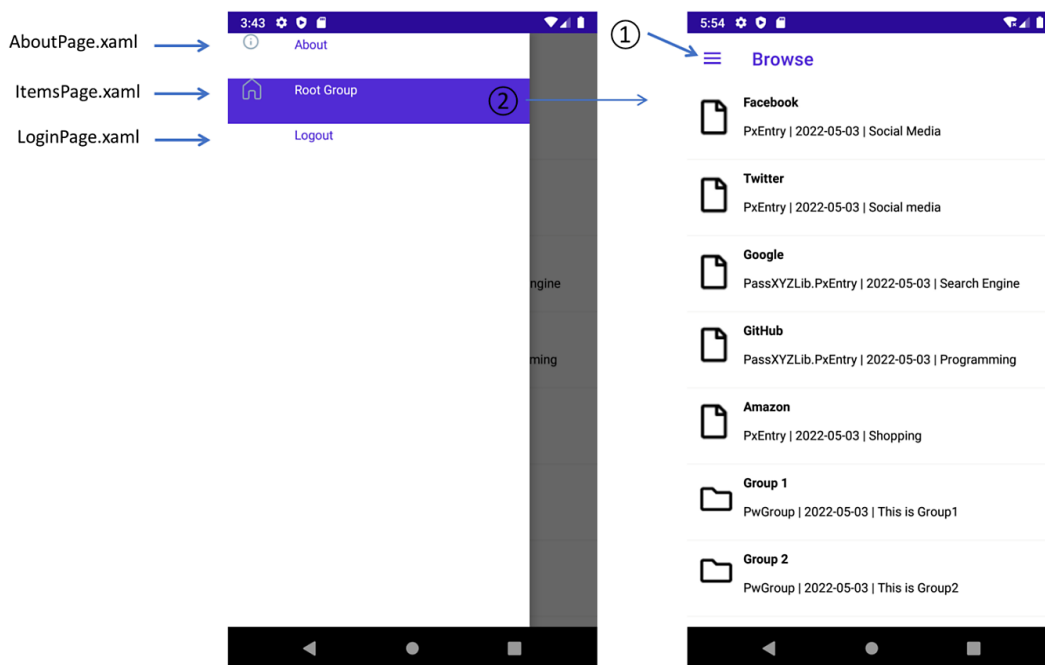


Figure 5.3: Flyout

The flyout menu consists of flyout items or menu items. In *Figure 5.3*, **About** and **Root Group** are flyout items, while **Logout** is a menu item.

Flyout items

Each flyout item is a `FlyoutItem` object that contains a `ShellContent` object. We can define flyout items like so in the `AppShell.xaml` file. We assign a string resource to the `Title` ① attribute and an `ImageSource` to the `Icon` ② attribute. These correspond to the properties of the `FlyoutItem` class:

```
<FlyoutItem ① Title="{x:Static resources:Resources.About}" ②
Icon="tab_info.png" >
  <Tab>
    <ShellContent Route="AboutPage" ContentTemplate=
      "{DataTemplate local:AboutPage}" />
  </Tab>
</FlyoutItem>
<FlyoutItem x:Name="RootItem" Title="Browse"
  Icon="tab_home.png">
  <Tab>
    <ShellContent Route="RootPage" ContentTemplate=
      "{DataTemplate local:ItemsPage}" />
  </Tab>
</FlyoutItem>
```

Shell has implicit conversion operators that can be used to remove the `FlyoutItem` and `Tab` objects so that the preceding XAML code can also be simplified, like so:

```
<ShellContent Title="{x:Static resources:Resources.
About}" Icon="tab_info.png" Route="AboutPage"
ContentTemplate="{DataTemplate local:AboutPage}" />
<ShellContent x:Name="RootItem" Title="Browse" Icon="tab_
home.png" Route="RootPage" ContentTemplate="{DataTemplate
local:ItemsPage}" />
```

Menu items

Flyout items can be used to navigate to a content page, but sometimes, we may want to take an action instead of navigating to a content page. In this case, we can use menu items. In our case, we have defined **Logout** as a menu item:

```
<MenuItem Text="Logout" IconImageSource="tab_login.png"
  Clicked="OnMenuItemClicked">
</MenuItem>
```

As we can see from the preceding XAML code, each menu item is a `MenuItem` object. The `MenuItem` class has a `Clicked` event and a `Command` property. When `MenuItem` is tapped, we can execute an action. In the preceding menu item, we assigned `OnMenuItemClicked()` as the event handler.

Let's review `AppShell.xaml` in our app in *Listing 5.1*. Here, we defined two flyout items and one menu item. We can select `AboutPage` ❶ and `ItemsPage` ❷ with flyout items and log out ❸ through the menu item:

Listing 5.1: `AppShell.xaml` in `PassXYZ.Vault` (<https://epa.ms/AppShell5-1>)

```
<Shell xmlns="http://schemas.microsoft.com
  /dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com
    /winfx/2009/xaml"
  xmlns:local="clr-namespace:PassXYZ.Vault.Views"
  xmlns:style="clr-namespace:PassXYZ.
    Vault.Resources.Styles"
  xmlns:resources="clr-namespace:PassXYZ.
    Vault.Properties"
  xmlns:app="clr-namespace:PassXYZ.Vault"
  Title="PassXYZ.Vault"
  x:Class="PassXYZ.Vault.AppShell">

  <Shell.Resources...>

  <TabBar>
    <Tab Title="{x:Static resources:Resources.
      action_id_login}" Icon="tab_login.png">
      <ShellContent Route="LoginPage" ContentTemplate=
        "{DataTemplate local:LoginPage}" />
    </Tab>
    <Tab Title="{x:Static resources:Resources.menu
      _id_users}" Icon="tab_users.png">
      <ShellContent Route="SignUpPage" ContentTemplate=
        "{DataTemplate local:SignUpPage}" />
    </Tab>
  </TabBar>
```

❹

```
<FlyoutItem Title="{x:Static resources:Resources.About}"
    Icon="tab_info.png" >
    <ShellContent Route="AboutPage" ContentTemplate=
        "{DataTemplate local:AboutPage}" /> ❶
</FlyoutItem>
<FlyoutItem x:Name="RootItem" Title="Browse"
    Icon="tab_home.png">
    <ShellContent Route="RootPage" ContentTemplate=
        "{DataTemplate local:ItemsPage}" /> ❷
</FlyoutItem>

<MenuItem Text="Logout" IconImageSource="tab_login.png"
    Clicked="OnMenuItemClicked"> ❸
</MenuItem>

</Shell>
```

There is also a **TabBar** ❹ defined for `LoginPage` and `SignUpPage`. Let's review tabs now.

Tabs

When we use tabs, `Shell` can create a navigation experience similar to `TabbedPage`. As shown in *Figure 5.4*, there are two tabs at the bottom tab bar on the Android and iOS platforms, but it looks different on the Windows platform:

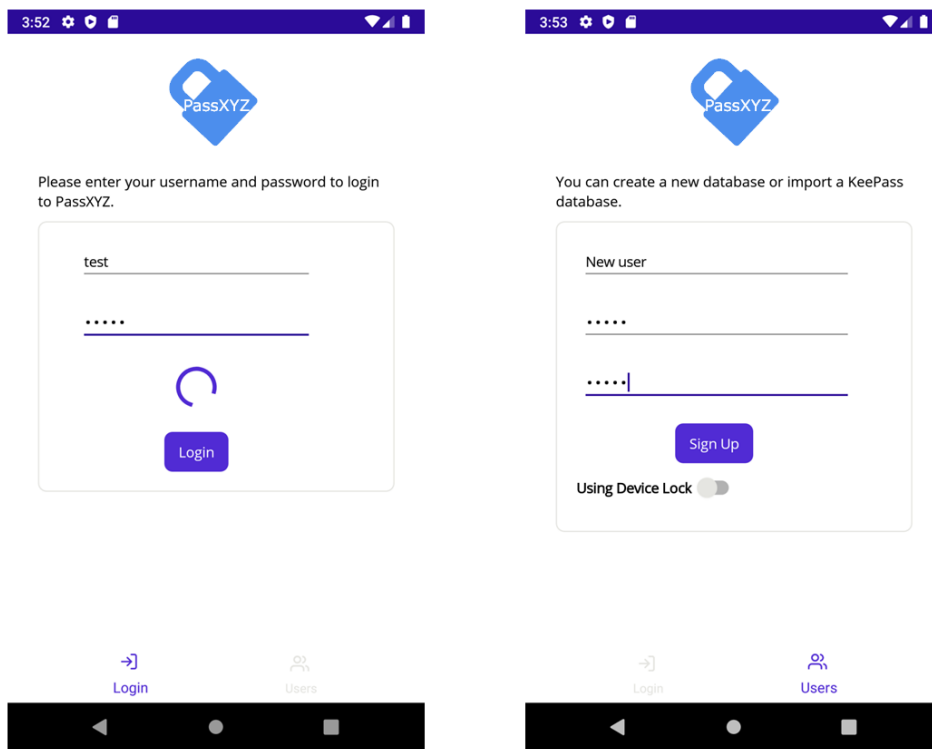


Figure 5.4: TabBar and tabs on Android

As we can see in *Figure 5.5*, on Windows, the tab bar is at the top:

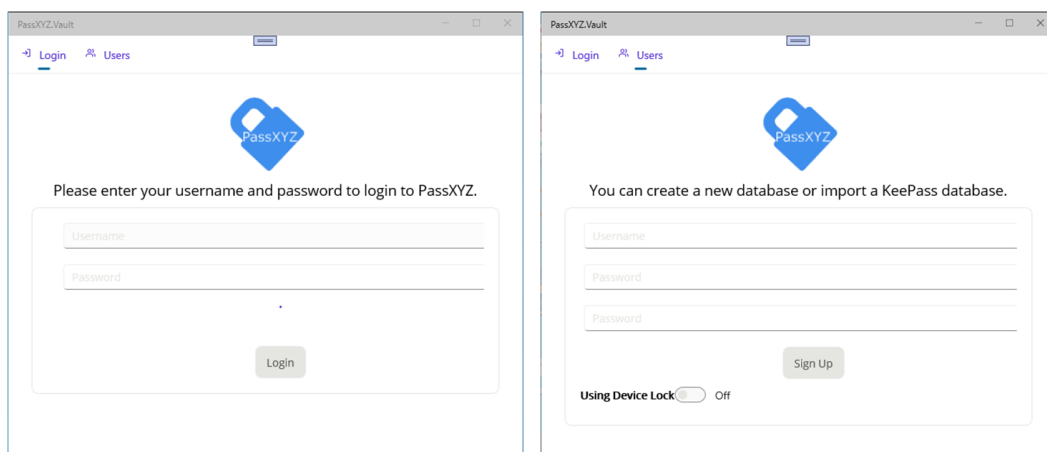


Figure 5.5: TabBar and tabs on Windows

To create tabs in our app, we must define a `TabBar` object. A `TabBar` object can contain one or more `Tab` objects and each `Tab` object represents a tab on the tab bar. Each `Tab` object can contain one or more `ShellContent` objects. The following XAML code shows that it is very similar to the one we get when we define a flyout:

```
<TabBar>
  <Tab Title="{x:Static resources:Resources.
    action_id_login}" Icon="tab_login.png">
    <ShellContent Route="LoginPage" ContentTemplate=
      "{DataTemplate local:LoginPage}" />
  </Tab>
  <Tab Title="{x:Static resources:Resources.menu_id_users}"
    Icon="tab_users.png">
    <ShellContent Route="SignUpPage" ContentTemplate=
      "{DataTemplate local:SignUpPage}" />
  </Tab>
</TabBar>
```

Just like what we did in the flyout XAML code, we can make the preceding code a little simpler by removing `Tab` tags. We can use the implicit conversion operators of `Shell` to remove `Tab` objects. As we can see, we can remove `Tab` tags and define `Title` and `Icon` attributes in `ShellContent` tags:

```
<TabBar>
  <ShellContent Title="{x:Static resources:Resources.
    action_id_login}" Icon="tab_login.png"
    Route="LoginPage" ContentTemplate="{DataTemplate
      local:LoginPage}" />
  <ShellContent Title="{x:Static resources:Resources.
    menu_id_users}" Icon="tab_users.png"
    Route="SignUpPage" ContentTemplate="{DataTemplate
      local:SignUpPage}" />
</TabBar>
```

If we define both `TabBar` objects and `FlyoutItem` objects in `AppShell.xaml`, `TabBar` disables the flyout items. That's the reason why, when we start our app, we can see a screen of tabs showing login or signup pages. After the user logs in successfully, we can bring the user to `RootPage`, which is the registered route in *Listing 5.1*. We'll learn how to register routes and navigate using registered routes in the next section.

Shell navigation

In Shell, we can navigate to pages through registered routes. There are two ways to register routes. The first way is to register routes in Shell's visual hierarchy. The second way is to register them explicitly using the `RegisterRoute()` static method of the `Routing` class.

Registering absolute routes

We can register routes in Shell's visual hierarchy as we did in *Listing 5.1*. We can specify a route through the `Route` property of `FlyoutItem`, `TabBar`, `Tab`, or `ShellContent`. In `AppShell.xaml`, we registered the following routes:

Route	Page	Description
LoginPage	LoginPage	This route displays a page for user login
SignUpPage	SignUpPage	This route displays a page for user signup
AboutPage	AboutPage	This route displays a page about our app
RootPage	ItemsPage	This route displays a page for navigating the password database

Table 5.1: Registered routes in the visual hierarchy

To navigate to a route in Shell's visual hierarchy, we can use an absolute route URI, such as `// LoginPage`.

Registering relative routes

We can also navigate to a page without pre-defining it in the visual hierarchy. For example, we can navigate to the password detail page, `ItemDetailPage`, at any hierarchy level of the password groups. In our app, we can register the following routes explicitly using `RegisterRoute()` in `App.xaml.cs`:

```
public App()
{
    InitializeComponent();
    Routing.RegisterRoute(nameof(ItemsPage),
        typeof(ItemsPage));
    Routing.RegisterRoute(nameof(ItemDetailPage),
        typeof(ItemDetailPage));
    Routing.RegisterRoute(nameof(NewItemPage),
        typeof(NewItemPage));
}
```

```
DependencyService.Register<MockDataStore>();  
DependencyService.Register<UserService>();  
    MainPage = new AppShell();  
}
```

In the preceding code, we defined the following routes:

Route	Page	Description
ItemDetailPage	ItemDetailPage	This is the route to display details about a password entry
NewItemPage	NewItemPage	This is the route to add a new item (entry or group)
ItemsPage	ItemsPage	This is the route to display a page for navigating the password database

Table 5.2: Registered detail page routes

To demonstrate how to use relative routes, we will add a new item. When we want to add a new item, we can navigate to `NewItemPage` using a relative route, like so:

```
await Shell.Current.GoToAsync(nameof(NewItemPage));
```

In this case, the `NewItemPage` route is searched and if the route is found, the page will be displayed and pushed to the navigation stack. The navigation stack here is the same as the one when we explained basic navigation using the `INavigation` interface. When we define a relative route and navigate to it, we pass a string as the name of the route. To avoid typos, we can use the class name as the route name by using the `nameof` expression.

After we have filled in the information for the new item in `NewItemPage`, we may click the **Save** or **Cancel** button. In the event handler of the **Save** or **Cancel** button, we can navigate back to the previous page using the following code:

```
await Shell.Current.Navigation.PopModalAsync();
```

Alternatively, we can use the following code:

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

As you can see from the preceding code, there are two ways we can navigate back. The first one uses the `PopModalAsync()` method of the `INavigation` interface. Since `Shell` itself is a derived class of `Page`, it implements the `INavigation` interface through the inherited `Navigation` property. We can call the modal `PopModalAsync()` navigation method to navigate back. Here, `NewItemPage` is a modal page.

The second approach is that we can use the `GoToAsync()` method to navigate back. Since `NewItemPage` is a modal page, you may be wondering how we can differentiate whether a page is a modal page or not when we call `GoToAsync()`. In `Shell` navigation, this is defined through page presentation mode. The content page of `NewItemPage` is defined like so:

```
<?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="PassXYZ.Vault.Views.NewItemPage"
  Shell.PresentationMode="ModalAnimated" ❶
  Title="New Item">
  <ContentPage.Content...>
</ContentPage >
```

As we can see, the `Shell.PresentationMode ❶` property is defined on the content page. Depending on whether we want to use animation, we can set a different value for this property. For a normal content page, we can set it to `NotAnimated` or `Animated`. For a modal page, we can set it to `Modal`, `ModalAnimated`, or `ModalNotAnimated`. If we use the default value, `Animated` is set.

To navigate back, the `GoToAsync()` method is used with the route set to `...`. This is a similar mechanism to filesystem navigation or browser URL navigation. The relative route, `...`, means navigating back to the parent route. It can be combined with a route to navigate a page at the parent level, like so:

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

In *Table 5.1* and *Table 5.2*, you will see that `ItemsPage` is registered as both the absolute route `RootPage` and relative route `ItemsPage`. `ItemsPage` may contain password groups at different levels. At the top level, it is an absolute route, but it is a relative route at all other navigation hierarchy levels.

Passing data to pages

To further explain why we register `ItemsPage` as both absolute and relative routes, let's review the navigation hierarchy of our app, as shown in *Figure 5.6*:

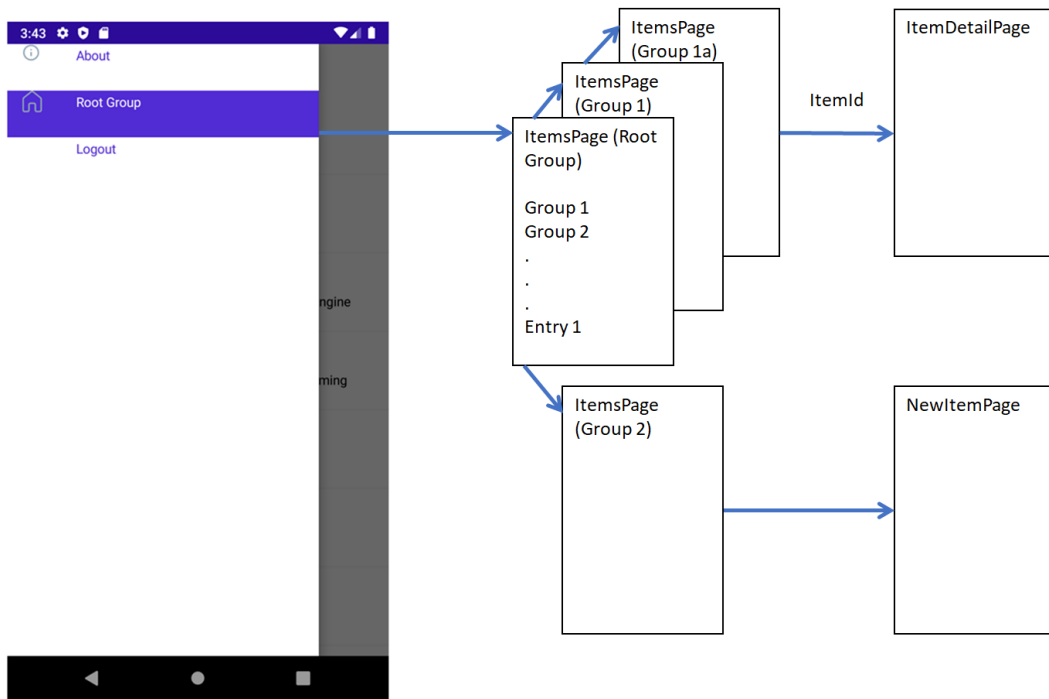


Figure 5.6: Navigation hierarchy

In our app, after successfully logging in, the main page displays a list of entries and groups at the top level of the password database called Root Group. This is similar to the navigation of the filesystem. At the root of the filesystem, the top level of files and folders are displayed.

The first instance of `ItemsPage` uses the `RootPage` route, which we can access through the flyout item. Let's say there are sub-groups called `Group1` and `Group2` in the `Root Group`, as shown in *Figure 5.6*. We can navigate to these sub-groups, which are instances of `ItemsPage`. These instances of `ItemsPage` cannot be pre-defined as they are relative routes and are pushed to the navigation stacks on demand. These navigation stacks can be as deep as the actual data stored in the password database.

These two different routes of `ItemsPage` are defined in `AppShell.xaml` and `App.xaml.cs` like so:

1. The `RootPage` route (absolute route):

```
<FlyoutItem x:Name="RootItem" Title="Browse" Icon="tab_
home.png">
  <ShellContent Route="RootPage" ContentTemplate=
    "{DataTemplate local:ItemsPage}" />
</FlyoutItem>
```

2. The `ItemsPage` route (relative route):

```
Routing.RegisterRoute (nameof (ItemsPage) ,
    typeof (ItemsPage) ) ;
```

Here, you may be wondering how we can navigate to `Group1` or `Group2` from `Root Group`. If `ItemsPage` can be used to display the content of either `Group1` or `Group2`, how can we tell `ItemsPage` which group to display?

In Shell navigation, data can be passed to a content page through query parameters. The syntax is similar to what we pass to a URL in the web browser. For example, we can use the following URL to search for `.net` in Google search: `https://www.google.com.hk/search?q=.net`.

This is achieved by appending `?` after a route with a pair of query parameter IDs and the value. In the preceding example, the key is `q` and the value is `.net`.

When an item in the list of `Root Groups` is selected, it can be an entry or a group. The click event triggers the `OnItemSelected()` method in `ItemsViewModel`, as shown in *Listing 5.2*:

Listing 5.2: `ItemsViewModel.cs` (<https://epa.ms/ItemsViewModel5-2>)

```
using PassXYZ.Vault.Views;
using System.Collections.ObjectModel;
using System.Diagnostics;
using KPCLib;
using PassXYZLib;
namespace PassXYZ.Vault.ViewModels;

[QueryProperty (nameof (ItemId) , nameof (ItemId))]
public class ItemsViewModel : BaseViewModel {
    private Item? _selectedItem = default;
    public ObservableCollection<Item> Items { get; }
    public Command LoadItemsCommand { get; }
    public Command AddItemCommand { get; }
    public Command<Item> ItemTapped { get; }

    public string ItemId {
        get {
            return _selectedItem == null ? string.Empty :
                _selectedItem.Id;
```

❶

❷

```

    }
    set {
        if (!string.IsNullOrEmpty(value)) {
            var item = DataStore.GetItem(value, true);
            if (item != null) {
                _selectedItem = DataStore.CurrentGroup =
                    DataStore.GetItem(value, true);
            }
            else {
                throw new ArgumentNullException("ItemId");
            }
        }
        else {
            _selectedItem = null;
            DataStore.CurrentGroup = DataStore.RootGroup;
        }
        ExecuteLoadItemsCommand();
    }
}

public ItemsViewModel()...
~ItemsViewModel()...
public async Task ExecuteLoadItemsCommand()...
async public void OnAppearing()...
public Item? SelectedItem...
private async void OnAddItem(object obj)...
public async void OnItemSelected(Item item) {
    if (item == null) return;
    if (item.IsGroup) {
        await Shell.Current.GoToAsync
            ($"{nameof(ItemsPage)}"?
            {nameof(ItemsViewModel.ItemId)}={item.Id}"); ❸
    }
    else {
        await Shell.Current.GoToAsync
            ($"{nameof(ItemDetailPage)}"?

```

```

        {nameof(ItemDetailViewModel.ItemId)}={item.Id}"); ❹
    }
}
}

```

According to the type of item, we may navigate to *ItemsPage* ❸ or *ItemDetailPage* ❹. In both cases, we pass the *Id* item to the *ItemId* query parameter, which is defined in both *ItemsViewModel* and *ItemDetailViewModel*.

In *Listing 5.2*, ❶ *ItemId* is defined in *ItemsViewModel* as the *QueryPropertyAttribute* attribute. The first argument of *QueryPropertyAttribute* is the name of the property that will receive the data. It is *ItemId* ❷ in this case.

The second argument is the *id* parameter. When we select a group from the list, the view model's *OnItemSelected()* method is invoked ❸ and the item *Id* of the selected group is passed as the value of the *ItemId* query parameter.

When *ItemsPage* is loaded with the *ItemId* query parameter, the *ItemId* ❷ property is set. In the setter of the *ItemId* property, we check whether the query parameter value is empty. If it is empty, it could be the first time we navigate to the *RootPage* route without a query parameter.

In this case, we set *CurrentGroup* of the data service to *RootGroup*. If it is not empty, we will find the item and set it to *CurrentGroup*. The content of *CurrentGroup* is loaded using the *ExecuteLoadItemsCommand()* method.

❹ If we select an entry from the list, we can navigate to *ItemDetailPage* with the item *Id* as the value of the query parameter. We can change *ItemDetailViewModel* like so to handle this query parameter:

```

public string? ItemId { ❶
    get {
        return itemId;
    }
    set {
        if (value == null) throw new ArgumentNullException
            (nameof(value));
        itemId = value;
        LoadItemId(value); ❷
    }
}

public ItemDetailViewModel() {

```

```
Fields = new ObservableCollection<Field>();
Id = default;
}

public async void LoadItemId(string itemId) {
    try {
        var item = await DataStore.GetItemAsync(itemId);    ③
        if (item == null) {
            throw new ArgumentNullException(nameof(itemId));
        }

        Id = item.Id;
        Title = item.Name;
        Description = item.Description;
        PwEntry dataEntry = (PwEntry)item;    ④
        Fields.Clear();
        List<Field> fields = dataEntry.GetFields(GetImage:
            FieldIcons.GetImage);    ⑤
        foreach (Field field in fields) {
            Fields.Add(field);
        }
    }
    catch (Exception) {
        Debug.WriteLine("Failed to Load Item");
    }
}
```

In the `ItemDetailViewModel` class, we have the following:

- `ItemId` ① is the property that receives the query parameter.
- In the setter of `ItemId`, we call the `LoadItemId()` method ② to load the item.
- In `LoadItemId()`, we can call the data service `GetItemAsync()` method ③ to get the item using the item `Id`.
- Here, the item is an instance of `PwEntry` ④, so we can cast it as a `PwEntry`.
- We have an extension method called `GetFields()` ⑤ in `PassXYZLib`. We use this method to update the list of fields.

We learned about basic navigation and `Shell` navigation in the last two sections. We also improved our navigation design using `Shell`. Now, it's time to review the MVVM pattern and refine our data model again to make our password manager app better.

Improving our model

We studied use cases and created some in *Chapter 4, Exploring MVVM and Data Binding*. In this section, we will enhance existing use cases and implement new use cases using the knowledge that we've learned.

We will work on the following use cases.

Use Case 1: As a password manager user, I want to log in to the password manager app so that I can access my password data.

For this use case, we haven't fully implemented the user login yet; we will complete this in the next chapter. In this chapter, we will implement some pseudo logic that includes everything except the data layer.

In *Chapter 4, Exploring MVVM and Data Binding*, we have the following use case, which can support one level of navigation.

Use case 3: As a password manager user, I want to see a list of groups and entries so that I can explore my password data.

To support multiple levels of navigation, we will implement the following use case in this section.

Use case 6: As a password manager user, when I click a group in the current list, I want to see the groups and entries belonging to this group.

Use case 7: As a password manager user, when I navigate my password data, I want to navigate back to the previous group or parent group.

In use cases 6 and 7, we want to navigate forward or backward using the relative routes.

In the MVVM pattern, we access our model through services. These services are usually abstracted as interfaces so that they're separate from the actual implementation. The `IDataStore` interface is one of them. To support use case 6 and improve use case 1, we need to create a new interface called `IUserService` to support user login.

Understanding the data model and its services

To understand the services and the enhanced model, let's review the enhanced design in *Figure 5.7*:

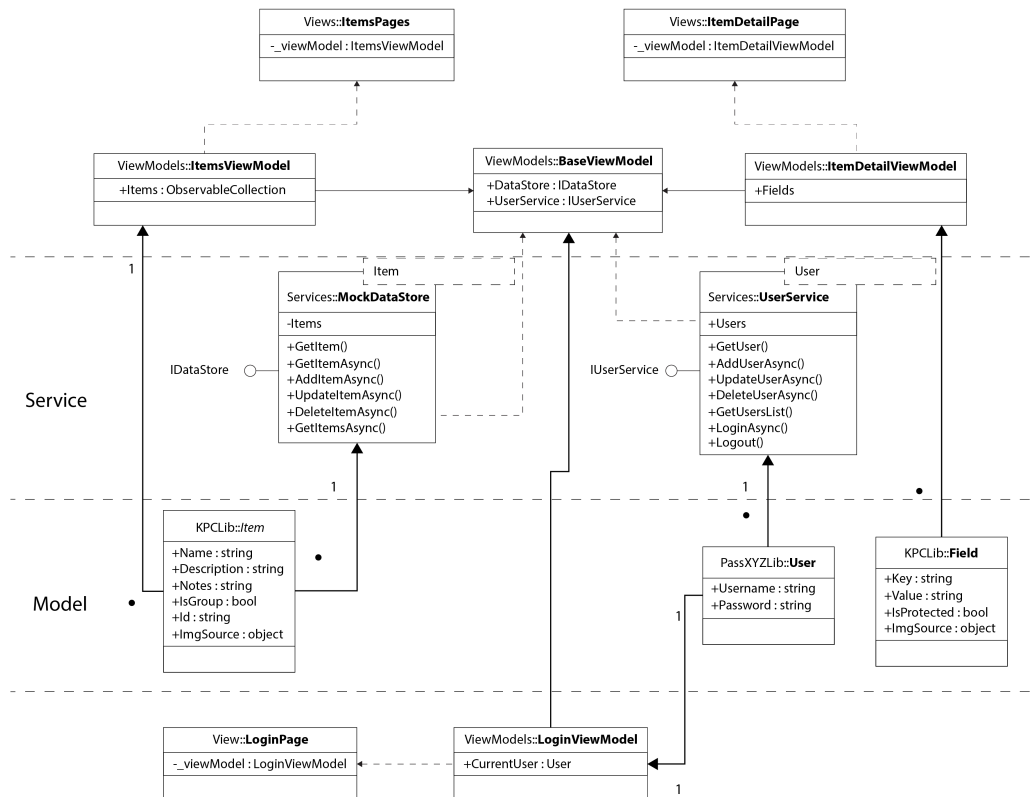


Figure 5.7: Model and service in MVVM

Figure 5.7 is a class diagram that depicts most of our design in the MVVM pattern. We can read this class diagram together with the following table to understand the MVVM pattern in our app. For simplicity, I have not included everything. For example, you can add `NewItemPage` or `SignUpPage` to *Figure 5.7* and *Table 5.3* by yourself:

Model		View	View Model
Data Model	Service		
User	IUserService	LoginPage	LoginViewModel
Item	IDataStore	ItemsPage	ItemsViewModel
Field		ItemDetailPage	ItemDetailViewModel

Table 5.3: Classes and interfaces in the MVVM pattern

To store application data, we usually store data in a database, which can be a relational database or NoSQL database. In our case, our password database is not a relational database. However, when we work on our design, we can still use the similar logic of relational databases to design our business logic. We have three classes to represent our model – `User`, `Item`, and `Field`.

`Item` and `Field` are used to represent a password entry and the content of an entry. We can imagine that an entry looks like a row in a table and that a `Field` is similar to a cell. We use `PwEntry` in `KeePassLib` to model a password entry. A list of entries is a group and we use `PwGroup` to model a group. Here, a group is similar to a table in a database. Fields with the same key values in a group are similar to a column. To design the interface of our data services, we can use a similar strategy to process data in our database.

How do we handle data in a database? You may have heard about CRUD operations. In our case, we can use the enhanced **Create, Read, Update, Delete, and List (CRUDL)** operations to define the interface of our service.

To process password entries and groups, we can use the `IDataStore` interface:

```
public interface IDataStore<T>
{
    T? GetItem(string id, bool SearchRecursive = false);
    Task<T?> GetItemAsync(string id, bool SearchRecursive =
        false);
    Task AddItemAsync(T item);
    Task UpdateItemAsync(T item);
    Task<bool> DeleteItemAsync(string id);
    Task<IEnumerable<T>> GetItemsAsync(bool forceRefresh =
        false);
}
```

In the `IDataStore` interface, we define the following CRUDL operations:

- **Create:** We use `AddItemAsync()` to add an entry or a group
- **Read:** We use `GetItem()` or `GetItemAsync()` to read an entry or a group
- **Update:** We use `UpdateItemAsync()` to update an entry or a group
- **Delete:** We use `DeleteItemAsync()` to delete an entry or a group
- **List:** We use `GetItemsAsync()` to get a list of entries and groups in the current group

To process users, we can use the `IUserService` interface:

```
public interface IUserService<T>
{
    T GetUser(string username);
    Task AddUserAsync(T user);
    Task UpdateUserAsync(T user);
    Task DeleteUserAsync(T user);
    List<string> GetUsersList();
    Task<bool> LoginAsync(T user);
    void Logout();
}
```

We can define a set of CRUDL operations to handle users as well:

- **Create:** We can create a new user using `AddUserAsync()`
- **Read:** We can get the user information using `GetUser()`
- **Update:** We can update a user using `UpdateUserAsync()`
- **Delete:** We can delete a user using `DeleteUserAsync()`
- **List:** We can get a list of users using `GetUsersList()`
- **Login and Logout:** We can log in or log out using an instance of `User`

To further separate the dependency of model and service, we can use generic types in the interface definition instead of concrete types. We use these services in our view models to manage our models. To improve the efficiency of our code, we will initialize the `IDataStore` service in `BaseViewModel` so that they are available in all derived view models automatically:

```
public class BaseViewModel : INotifyPropertyChanged {
    public static IDataStore<Item> DataStore =>
        DependencyService.Get<IDataStore<Item>>();

    bool isBusy = false;
    public bool IsBusy...
    string title = string.Empty;
    public string Title...

    ...
}
```

In the `BaseViewModel` class, we initialize the `IDataStore` service through a dependency service. We will explain dependency services and dependency injection in the next chapter.

Improving the login process

For user management, we may add new users or delete obsolete users in the system. We only have one user who can log in to our app at a time, so we must define a singleton class called `LoginUser` to model this case:

Listing 5.3: `LoginUser.cs` (<https://epa.ms/LoginUser5-3>)

```
using System.Diagnostics;
using PassXYZLib;
namespace PassXYZ.Vault.Services;

public class LoginUser : PxUser
{
    private const string PrivacyNotice = "Privacy Notice";
    public static bool IsPrivacyNoticeAccepted...
    private bool _isFingerprintEnabled = false;
    public bool IsFingerprintEnabled =>
        _isFingerprintEnabled;

    public static IUserService<User> UserService =>
        DependencyService.Get<IUserService<User>>(); ❶
    public override void Logout() {
        UserService.Logout();
    }

    public async Task<string> GetSecurityAsync()...
    public async Task SetSecurityAsync(string passwd)...
    public async Task<bool> DisableSecurityAsync()...

    private LoginUser() { } ❷
    private static LoginUser? instance = null;
    public static LoginUser Instance { ❸
        get {
            if (instance == null) { instance = new
```

```

        LoginUser(); }
        return instance;
    }
}
}
}
}

```

LoginUser is inherited from the User class through a sub-class called PxUser.

- ❶ In LoginUser, we initialize the IUserService interface through a dependency service.
- ❷ To implement a singleton class, we make the default constructor private to disable the creation of this class using a new operator.
- ❸ To get an instance of LoginUser, we must define a static property Instance. Please be aware that I chose to use a no-thread-safe implementation here to keep the code simple. In production implementation, we should use a lock to make it thread-safe.

Once we have implemented the IUserService interface and the LoginUser class, we can improve the login process. So far, we only have one Button on the login page. Let's add a username field and a password field to LoginPage.xaml, as shown in *Listing 5.4*:

Listing 5.4: LoginPage.xaml (<https://epa.ms/LoginPage5-4>)

```

<?xml version="1.0" encoding="utf-8" ?>
<ContentPage...>
    <ContentPage.Content>
        <StackLayout Padding="30" Spacing="10">
            <Image HorizontalOptions="Center"
                HeightRequest="96"...>
            <Label FontSize="Small".../>
            <Frame Margin="10">
                <Grid x:DataType="viewmodels:LoginViewModel"...>❶
                    <Grid.RowDefinitions...>
                    <Grid.ColumnDefinitions...>
                    <!-- Row 1 (Username) -->
                    <Image Grid.Row="0" Grid.Column="0"...>
                    <Entry x:Name="usernameEntry" Placeholder="{x:Static
resources:Resources.
    field_id_username}" ReturnType="Next" Text="{Binding
    CurrentUser.Username}" HorizontalOptions="Fill"
                        Grid.Row="0" Grid.Column="1" />❷
                
```

```

<ImageButton x:Name="switchUsersButton"...>
<!-- Row 2 (Password) -->
<Image Grid.Row="1" Grid.Column="0"...>
<Entry x:Name="passwordEntry" Placeholder="{x:Static
resources:Resources.
field_id_password}" IsPassword="true" Text="{Binding
CurrentUser.Password}" HorizontalOptions="Fill"
    Grid.Row="1" Grid.Column="1" /> ❸
<!-- Row 3 (ActivityIndicator) -->
<ActivityIndicator IsRunning="{Binding IsBusy}"
Grid.Row="2" Grid.Column="1" IsVisible="{Binding
    IsBusy}" /> ❹
<!-- Row 4 (Login Button) -->
<Button Text="{x:Static resources:Resources.
    LoginPageTitle}" HorizontalOptions=
    "CenterAndExpand" Command="{Binding
    LoginCommand}" Grid.Row="3" Grid.Column="1" /> ❺
</Grid>
</Frame>
<Label x:Name="messageLabel" />
</StackLayout>
</ContentPage.Content>
</ContentPage>

```

In this new user interface design, we did the following:

- ❶ We created a 4x3 grid layout in a frame.
- In the first two rows, we used two instances of `Entry` to hold the username ❷ and password ❸. We created a data binding between the `Text` fields of `Entry` and the properties of `CurrentUser`, which is defined in the view model. It is an object of `LoginUser`.
- ❹ In the third row, we used an `ActivityIndicator` control to show the login status, which is bound to the `IsBusy` property of the view model.
- ❺ In the last row, we defined a `Button` control for the login activity. There is a `Command` property defined in `Button` that implements the `ICommand` interface. We used data binding to link this `Command` property to the method in the view model to perform the login activity.

We can see the improved login user interface in *Figure 5.8*:

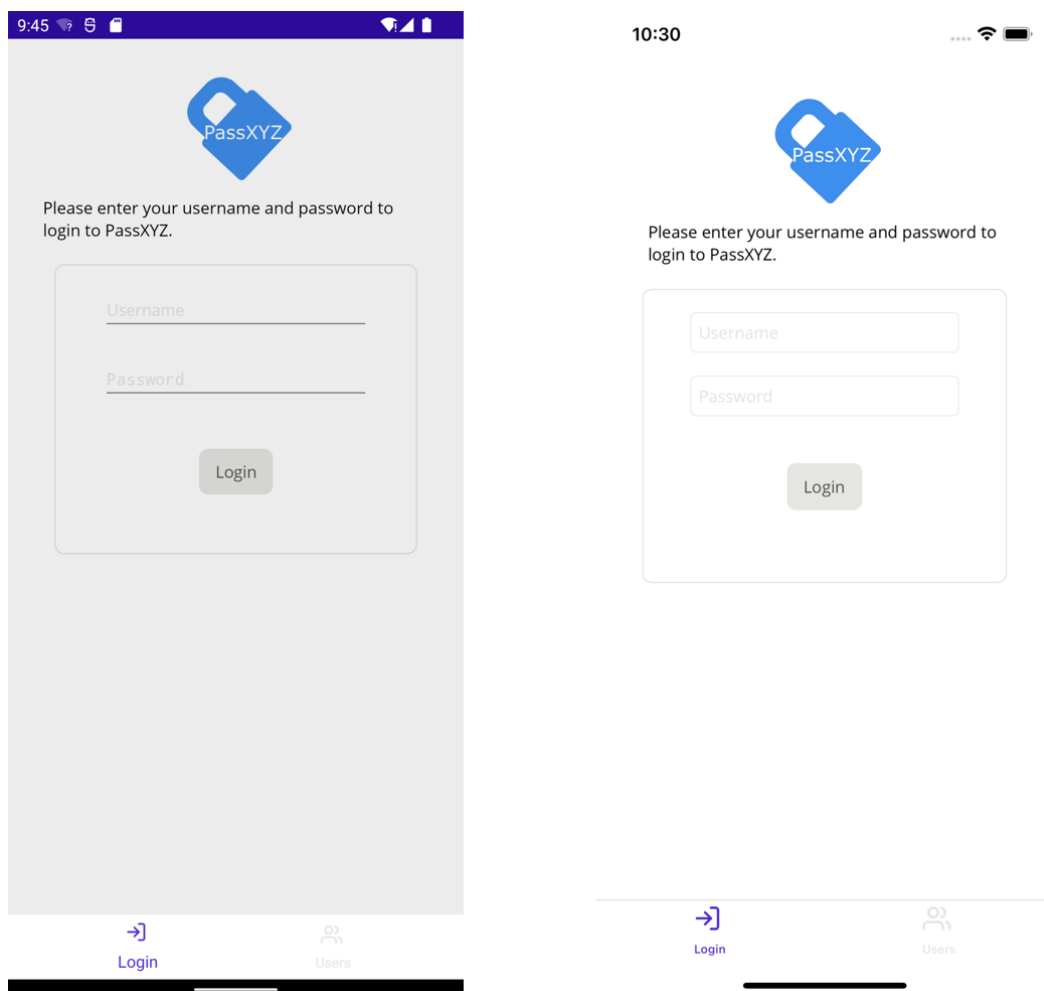


Figure 5.8: LoginPage

The Command interface

Without Command property support, we must create an event handler of the Clicked event in the code-behind file of LoginPage. In the event handler, we call a method defined in the view model that invokes `LoginAsync()` in `IUserService` to process login activity. Using the Command property, we can bind to the `LoginCommand` property in the view model directly, so we don't need to create the event handler in the code-behind file of LoginPage. The code looks cleaner and simpler.

Let's review the improved `LoginViewModel.cs` file to find out more details:

Listing 5.5: LoginViewModel.cs (<https://epa.ms/LoginViewModel5-5>)

```
public class LoginViewModel : BaseViewModel
{
    readonly IUserService<User> userService = LoginUser.
UserService;
    private Action<string> _signUpAction;
    public Command LoginCommand { get; }
    public Command SignUpCommand { get; }
    public Command CancelCommand { get; }
    public LoginUser CurrentUser => LoginUser.Instance;
    public ObservableCollection<User>? Users ...

    public LoginViewModel() {
        LoginCommand = new Command(OnLoginClicked,
            ValidateLogin); ❶
        SignUpCommand = new Command(OnSignUpClicked,
            ValidateSignUp);
        CancelCommand = new Command(OnCancelClicked);
        CurrentUser.PropertyChanged +=
            (_, __) => LoginCommand.ChangeCanExecute(); ❷
        CurrentUser.PropertyChanged +=
            (_, __) => SignUpCommand.ChangeCanExecute();
    }
    public LoginViewModel(Action<string> signUpAction) ...
    private bool ValidateLogin() { ❸
        return !string.IsNullOrWhiteSpace(CurrentUser.Username)
            && !string.IsNullOrWhiteSpace(CurrentUser.Password);
    }
    private bool ValidateSignUp() ...
    public void OnAppearing() ...
    public async void OnLoginClicked() {
        try {
            IsBusy = true;
            if (string.IsNullOrWhiteSpace
```

```

        (CurrentUser.Password))...
    bool status = await userService.LoginAsync
        (CurrentUser);
    if (status) {
        if (AppShell.CurrentAppShell != null) {
            AppShell.CurrentAppShell.SetRootPageTitle
                (DataStore.RootGroup.Name);
            string path = Path.Combine
                (PxDataFile.TmpFilePath,
                 CurrentUser.FileName);
            if (File.Exists(path))...
            await Shell.Current.GoToAsync($"//RootPage");
        }
        else {
            throw (new NullReferenceException
                ("CurrentAppShell is null"));
        }
    }
    IsBusy = false;
}
catch (Exception ex) {
    IsBusy = false;
    string msg = ex.Message;
    if (ex is System.IO.IOException ioException) {
        msg = Properties.Resources.message_
            id_recover_datafile;
    }
    await Shell.Current.DisplayAlert
        (Properties.Resources.LoginErrorMessage, msg,
         Properties.Resources.alert_id_ok);
}
}
}

```

4

In `LoginViewModel` (Listing 5.5), we defined a few Command properties. Let's look at `LoginCommand` to understand how to use the `ICommand` interface.

In .NET MAUI, the `Command` class is defined like so:

```
public class Command : System.Windows.Input.ICommand
```

❶ We use the following constructor to initialize `LoginCommand`:

```
public Command (Action execute, Func<bool> canExecute);
```

The `execute` parameter, which is of the `Action` type, is the action to be invoked. Here, the `OnLoginClicked()` method is assigned. When the user clicks the button, it will be executed.

We also assigned another method called `ValidateLogin()` ❸ to the `canExecute` parameter. This parameter is used to indicate whether this `Command` can be executed. In the `ValidateLogin()` method, we check whether the username or password in `CurrentUser` is empty. If either of them is empty, this `Command` cannot be executed, and the button should be grayed out.

To detect a change in the username or password, we can register `ChangeCanExecute()` to the `PropertyChanged` handler ❷.

After the user clicks the button, the `OnLoginClicked()` method is called. In this method, we pass `CurrentUser` to the `IUserService` method's `LoginAsync()` ❹ to handle the login process.

So far, we have improved our model and service to enhance the login process. If we recall the class diagram in *Figure 5.7*, we have changed the source code at the view, view model, and service layers to enhance our app. The actual model is encapsulated in two libraries called `KPCLib` and `PassXYZLib`. We exposed the functionalities of these two libraries through the `IDataStore` and `IUserService` interfaces. We enhanced our model by building the actual implementation classes of these two interfaces. In the next chapter, we will continue improving these two service interfaces and build a fully functional app.

Summary

In this chapter, we learned about basic navigation and `Shell`. We used `Shell` as the navigation framework in our app design. We explored the features of `Shell` and explained how to use it in our app design.

After we completed most of the user interface design, we enhanced our model by making changes to two service interfaces: `IDataStore` and `IUserService`. We improved the login process after making changes in the view, view model, and service layers. In the service layer, we are still using the `MockDataStore` class. However, we haven't finalized the implementation in the `IDataStore` service to perform the actual login activities yet. We will leave this to the next chapter.

In the next chapter, we will explain dependency injection in .NET MAUI, which is a major difference compared to Xamarin.Forms. We will learn how to register our services via dependency injection and how to initialize our service through constructor injection or property injection. We will also create the actual service to replace `MockDataStore`.

6

Introducing Dependency Injection and Platform-Specific Services

In the last chapter, we introduced navigation and Shell in **.NET Multi-platform App UI (.NET MAUI)**, and we completed the navigation design of our app. We improved two interfaces, `IDataStore` and `IUserService`, to separate the model from the view and view model. In the current code, we used the `DependencyService` class to decouple the interface implementations. In this chapter, we will refine our design using **dependency injection (DI)** to replace the `DependencyService` class. There is a built-in service to support DI in .NET MAUI. With the help of DI, we can refine our design and decouple the dependencies in a more elegant way.

We will cover the following topics in this chapter:

- A quick review of design principles
- Using DI
- Connecting to the database

DI is a technique to realize the design principle of dependency inversion or **Dependency Inversion Principle (DIP)**. DIP is one of the SOLID design principles, and we will learn how to use SOLID design principles in our design. We will have a quick overview of SOLID design principles at the beginning of this chapter before we dive into DI.

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for further details.

The source code for this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter06>

The source code can be downloaded using the following Git command:

```
git clone -b chapter06 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

A quick review of design principles

Design principles are high-level guidelines about design considerations. They can give fundamental pieces of advice for you to make a better design decision. There are general design principles that can be used not only for software design but are also applicable to other design works.

Let's review some general design principles before we move to commonly used design principles (SOLID) in software development.

Exploring types of design principles

Design principles can be a huge topic. So, instead of a detailed description, here, I will share my experience of applying design principles in development by giving you a quick review of the design principles used in this book. We will start with high-level principles such as **DRY**, **KISS**, **YAGNI**, and so on first, and then we move to the ones that are mostly used in software development. The most commonly used ones in **object-oriented programming (OOP)** are SOLID design principles.

Don't Repeat Yourself (DRY)

As people often say, don't reinvent the wheel; we should try to reuse existing components rather than redevelop something that already exists.

Keep It Simple, Stupid (KISS)

We should choose a simple and straightforward approach rather than involve unnecessary complexity in a design.

You Aren't Gonna Need It (YAGNI)

We should implement functionality when it is required. In software development, there is a tendency to futureproof a design. This may create something that is actually not needed and increase the complexity of the solution.

SOLID design principles

SOLID design principles are the ones used in software development. They are high-level guidelines for many design patterns. SOLID is an acronym for the following five principles:

- **Single Responsibility Principle (SRP)**—A class should only have one responsibility. As a developer, you have one and only one reason to change a class. With this design principle in mind, the implementation is easier to understand and more efficient to deal with as requirements change.
- **Open/Closed Principle (OCP)**—Classes should be open for extension but closed for modification. The main idea of this principle is to keep the existing code from breaking when you implement new features.
- **Liskov Substitution Principle (LSP)**—If the object of the parent type can be used in a context, the object of the child type should be able to be used the same way without anything breaking.
- **Interface Segregation Principle (ISP)**—A design should not implement an interface that it doesn't use, and a class should not be forced to depend on methods it doesn't intend to implement. We should design concise and simple interfaces rather than large and complex ones.
- **Dependency Inversion Principle (DIP)**—This principle emphasizes the decoupling of software modules. High-level modules should not depend on low-level modules directly. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Design principles are guidelines to help us to make better design decisions. However, it is up to us to decide what to do in the actual implementation, not the design principles.

Using design principles

Now that we've talked about the different design principles, let me share my lessons learned when using design principles.

In the model of our app, I reused `KeepassLib` from Dominik Reichl. When I ported it to .NET Standard, I changed the inheritance hierarchy, as shown in *Figure 6.1*:

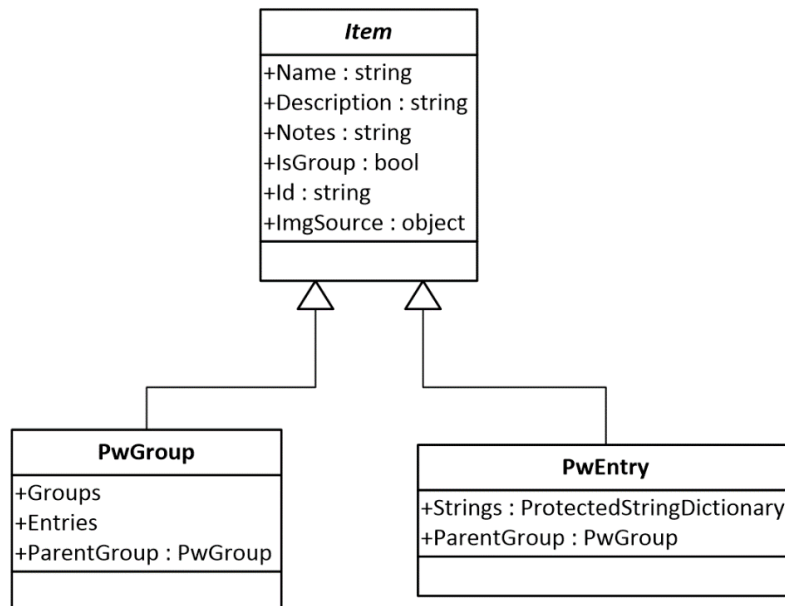


Figure 6.1: Class diagram of Item, PwEntry, and PwGroup

I created an abstract parent class, `Item`, for the group (`PwGroup`) and entry (`PwEntry`). It looks like this change breaks OCP in SOLID. The reason that I did it this way is influenced by a lesson I learned in the previous implementation.

So, previously I did not implement `KPCLib` this way before *version 1.2.3*. At that time, I used `PwGroup` and `PwEntry` directly, so I had to handle groups and entries separately. You can imagine the increased complexity in `ItemsPage` and `ItemsViewModel`. The most important side effect is that I couldn't separate the model and view model clearly. In the view model, I had to handle a lot of details using `KeePassLib` directly. After the `Item` abstract parent class is introduced, I can hide most of the detailed implementation in services (`IDataStore` and `IUserService`) and `PassXYZLib`. No code that is dependent on `KeePassLib` is present in the view and view model anymore. The thought behind this change is influenced more by KISS rather than just sticking to OCP. The result is that the overall architecture looks much better if we consider other SOLID principles, such as LSP and SRP. The point that I want to share here is that we may find conflicts among various design principles in the actual work. It's up to us to make decisions rather than just sticking to design principles. The best design decision usually comes from the lessons learned in previous failure cases.

Now, coming back to our primary topic, we'll talk about refining design using one of the SOLID principles—dependency inversion. In SOLID design principles, dependency inversion emphasizes the decoupling of software modules, and it also gives guidelines about how to do it. The key idea behind it is we should try to depend on abstractions whenever possible. In the actual implementation, DI is the technique that we use frequently to apply the idea of dependency inversion.

Using DI

DI is one of the tools that we can use in .NET MAUI. It is actually not something new, and it has been used heavily in backend frameworks such as ASP.NET Core or the Java Spring Framework. DI is a technique for achieving dependency inversion (DIP). It can help to decouple the usage of an object from its creation so that we don't have to depend on the object we use directly. In our app, after we decouple the implementation of the `IDataStore` interface, we can start with a mock implementation first and replace it with the actual implementation later.

In .NET MAUI, `Microsoft.Extensions.DependencyInjection`—or MS.DI, as we shorten the name in this chapter—is a built-in service that we can use directly.

In the .NET world, there are many DI containers available other than MS.DI. They may be more powerful and flexible than MS.DI, such as the Autofac DI container or the Simple Injector DI container, and so on. Then, why do we choose MS.DI instead of other powerful and flexible DI containers? Here, we may want to think about KISS and YAGNI principles again. We should not choose a more powerful solution by assuming that we may use some features in the future. The simplest and easiest solution is to use what we already have without any extra effect. With MS.DI, we don't have to involve any extra dependencies. Irrespective of whether we want to use it or not, it is already there in the configuration of .NET MAUI. We can just add a few lines of code to make our design better. For other DI containers, it may be better at imagining the future, but we have to introduce additional dependencies and do the necessary configuration in our code before we can really use them. If you are working on a complex system design, you may want to evaluate the available DI containers and choose the right one for your system. For our case, `PassXYZ.Vault` is a relatively simple app, and we won't benefit directly from the advanced DI features provided by Autofac or Simple Injector. The functionalities provided by MS.DI are sufficient for our implementation.

Before we move to the topic of DI, let's look at our current implementation first. Instead of using DI, we are using **DependencyService** from `Xamarin.Forms` in our app to decouple the interface and its implementation. We will refine our code to replace `DependencyService` with DI in this chapter.

Dependency Service

In *Chapter 2, Building Our First .NET MAUI App*, we created our app from a `Xamarin.Forms` template and migrated it to .NET MAUI. Whatever works in `Xamarin.Forms` can still work in .NET MAUI, including `DependencyService`.

`DependencyService` is a service locator that enables `Xamarin.Forms` applications to invoke native functionality from shared code, but it can also be used to play a DI container role for simple use cases.

The module that we want to decouple in our app is the model layer, which is a third-party library from KeePass. As we can see in the package diagram in *Figure 6.2*, our system includes three separate assemblies: `KPCLib`, `PassXYZLib`, and `PassXYZ.Vault`:

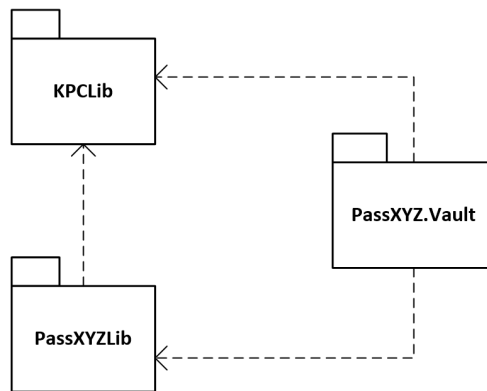


Figure 6.2: Package diagram

The **KPCLib** package includes two namespaces, **KeePassLib** and **KPCLib**. **PassXYZLib** is a package to extend the functionality of the **KPCLib** package with .NET MAUI-specific implementation. **PassXYZ.Vault** is our app that depends on the **PassXYZLib** package directly and depends on the **KPCLib** package indirectly. According to the DI principle, we want to create dependencies on abstractions rather than actual implementations. We created two interfaces, **IDataStore** and **IUserService**, which we can use to decouple from the actual implementations.

Using **DependencyService** includes two steps—registration and resolution. Let's look at this in more detail:

1. Registering a service

We need to register the **IDataStore** and **IUserService** interfaces first before we can use them. In the current code, they are registered in the constructor of the **App** class, as shown here:

```
DependencyService.Register<MockDataStore>();
DependencyService.Register<UserService>();
```

We can use the **Register()** method of **DependencyService** to register the implementation of services. The **MockDataStore** and **UserService** classes implement **IDataStore** and **IUserService** interfaces. **MockDataStore** is not the actual implementation, and it is used for testing purposes only. We will replace it with the actual implementation later in this chapter. This is one of the benefits that we can see after we decouple from the actual implementation.

2. Resolving a service

To resolve the dependency in our code, we defined a **DataStore** public property of type **IDataStore** in the **BaseViewModel** class, as shown here:

```
public static IDataStore<Item> DataStore =>
    DependencyService.Get<IDataStore<Item>>();
```

We can use the `Get()` method of `DependencyService` to resolve the dependency. Since `BaseViewModel` is the parent class of all other view models, we can access the `IDataStore` interface in all view models with this setup.

For the `IUserService` interface, we created a `LoginUser` singleton class and defined a `UserService` public property in the `LoginUser` class, as shown here:

```
public static IUserService<User> UserService =>
    DependencyService.Get<IUserService<User>>();
```

This is the current implementation of `DependencyService` in our app. Let's replace it with DI supported in MS.DI.

Using built-in MS.DI DI service

To use MS.DI as a DI service, the usage is similar to `DependencyService`, which includes two steps—registration and resolution. We can use the `ServiceCollection` class for the registration and the `ServiceProvider` class for the resolution, as shown in *Figure 6.3*:

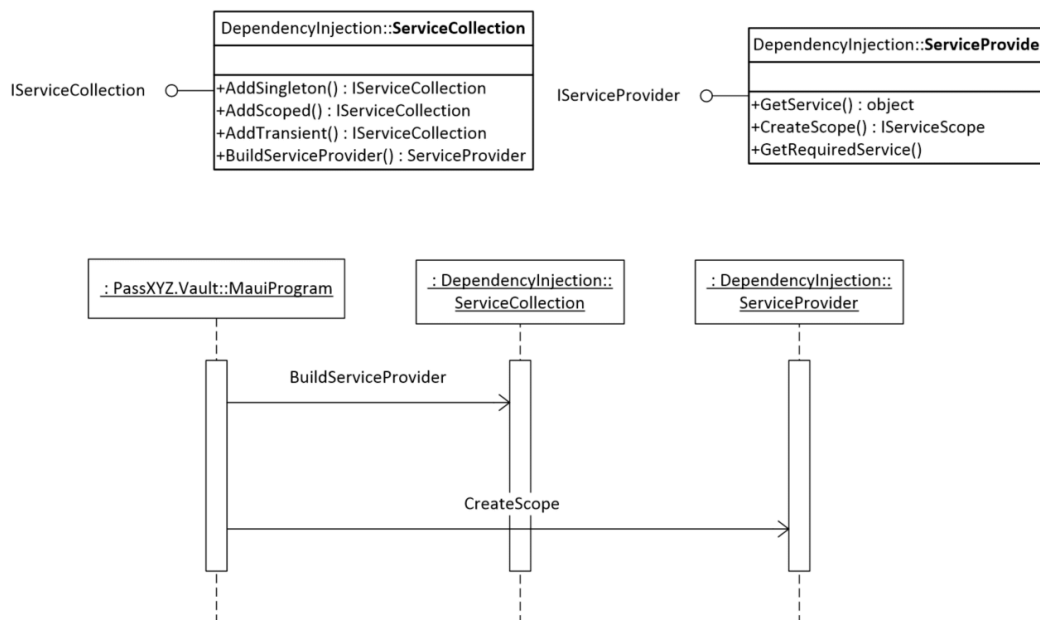


Figure 6.3: Usage of MS.DI

If we want to use DI to resolve the `IDataStore` service, we can use the steps in the following code block:

```
// Registration
var services = new ServiceCollection();           ❶
services.AddSingleton<IDataStore<Item>, MockDataStore>(); ❷
// Resolution
ServiceProvider provider =
    services.BuildServiceProvider(validateScopes: true); ❸
IDataStore<Item> dataStore =
    provider.GetRequiredService<IDataStore<Item>>();      ❹
```

❶ We need to create an instance of the `ServiceCollection` class first. `ServiceCollection` implements the `IServiceCollection` interface.

❷ The `IServiceCollection` interface itself does not define any method directly. There are a set of extension methods defined in MS.DI. We can use the `AddSingleton()` extension method to register the concrete `MockDataStore` class for the `IDataStore` abstraction. The `AddSingleton()` method can use a generic type to define the interface and the implementation. There are multiple overloaded versions of the `AddSingleton()` extension method available.

❸ To resolve objects, we can get an instance of `ServiceProvider` by calling the `BuildServiceProvider()` extension method of `IServiceCollection`. The `ServiceProvider` class implements the `IServiceProvider` interface. The `IServiceProvider` interface is defined in the `System` namespace and it defines only one method, `GetService()`. The rest of the methods are extension methods defined in the `Microsoft.Extensions.DependencyInjection` namespace, as we can see in *Figure 6.3*.

❹ Once we have an instance of `ServiceProvider`, we can resolve the `IDataStore` interface using the `GetRequiredService()` extension method.

Even though MS.DI is a lightweight DI service, it provides enough features for .NET MAUI applications, as set out here:

- Lifetime management of instances
- Constructor, method, and property injections

Let's explore these features in the next sections.

Lifetime management

We can manage the lifetime of instances by configuring `ServiceCollection`.

To configure `ServiceCollection`, we can use the following three extension methods:

- `AddSingleton`—This method creates a single instance throughout the life of the application. It creates an instance for the first time and reuses it in the following calls.
- `AddTransient`—This method creates an instance for each call. The lifetime of the instance depends on the scope of the programming logic.
- `AddScoped`—The lifetime of the instance resolved by this method is within the scope defined by the design. It creates one instance and reuses the same instance within the defined scope. In ASP.NET, we can define the scope as the session of each HTTP request.

To explain the lifetime management of MS.DI, we can review the following code snippet, together with *Figure 6.4*:

```
var services = new ServiceCollection();
services.AddSingleton< IUserService<User>, UserService>(); ❶
services.AddScoped<IDataStore<Item>, DataStore>();          ❶
services.AddTransient<ItemsViewModel>();                   ❶
ServiceProvider rootContainer =
    services.BuildServiceProvider(validateScopes: true);    ❷

var userService =
    rootContainer.GetRequiredService<IUserService<User>>();

IServiceScope scope1 = rootContainer.CreateScope();         ❸
IDataStore<Item> dataStore1 =
    scope1.ServiceProvider.GetRequiredService<IDataStore<Item>>
    ();

IServiceScope scope2 = rootContainer.CreateScope();         ❸
IDataStore<Item> dataStore2 =    Scope2.ServiceProvider.
    GetRequiredService<IDataStore<Item>>
    ();
```

In the preceding code, ❶ we registered `IUserService` as a Singleton object, `IDataStore` as a Scoped object, and `ItemsViewModel` as a Transient object.

After the registration, ❷ we created a `ServiceProvider` instance and stored it in a `rootContainer` variable. ❸ We created two scopes, `scope1` and `scope2`, using the `rootContainer`. We can review the lifetime management of the created objects in *Figure 6.4*:

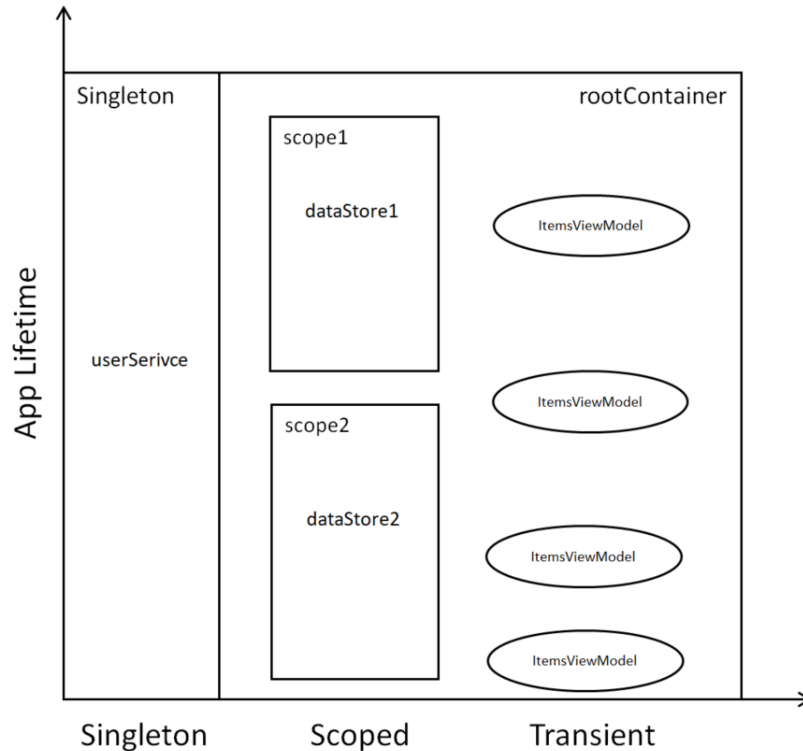


Figure 6.4: Lifetime management in MS.DI

`userService` is created as a `Singleton` object, so there is only one instance, and the instance has the same lifetime as the application itself. The two scopes that are `scope1` and `scope2` have their own lifetimes that are decided by our design. The `Scoped` objects, `dataStore1` and `dataStore2`, have the same lifetime as the scope to which they belong. The instances of `ItemsViewModel` are `Transient` objects.

For each of these three methods, `AddSingleton()`, `AddScoped()`, and `AddTransient()`, multiple overloaded versions are defined to meet various requirements in the `ServiceCollection` configuration.

In our application, we have two versions of the `IDataStore` interface implementation:

1. `DataStore`—This is the actual implementation
2. `MockDataStore`—This is the one used for testing purposes

Using MS.DI, we can use `MockDataStore` in the Debug build and use `DataStore` in the Release build. This configuration can be implemented as shown in the following code snippet:

```
bool isDebug = false;
var services = new ServiceCollection();
services.AddSingleton<DataStore, DataStore>();
services.AddSingleton<MockDataStore, MockDataStore>();
services.AddSingleton<IDataStore<Item>>(c => {
    if (isDebug)
    {
        return c.GetRequiredService<MockDataStore>();
    }
    else
    {
        return c.GetRequiredService<DataStore>();
    }
});
```

In the preceding code snippet, we can configure concrete classes and `DataStore`, `MockDataStore`, and `IDataStore` interfaces for different build configurations. In the configuration of `IDataStore`, we can use a delegate to resolve the object. The `isDebug` variable can be set using the build configuration so that it can be set to `true/false` according to whether it is a debug or release build.

Configuring DI in .NET MAUI

MS.DI is implemented as part of the .NET release, so it is available for all kinds of applications in .NET 5 or later releases. We can implement DI using `ServiceCollection` and `ServiceProvider`, as we introduced in the previous section. However, there is a much simpler way to use MS.DI in .NET MAUI. DI is integrated as part of the .NET Generic Host configuration, so we don't need to create an instance of `ServiceCollection` by ourselves. We can use the preconfigured DI service directly without any extra work.

To understand the preconfigured DI service in .NET MAUI, we can review the .NET MAUI application startup process again using *Figure 6.5*. *Figure 6.5* includes both a class diagram and a sequence diagram of the involved classes:

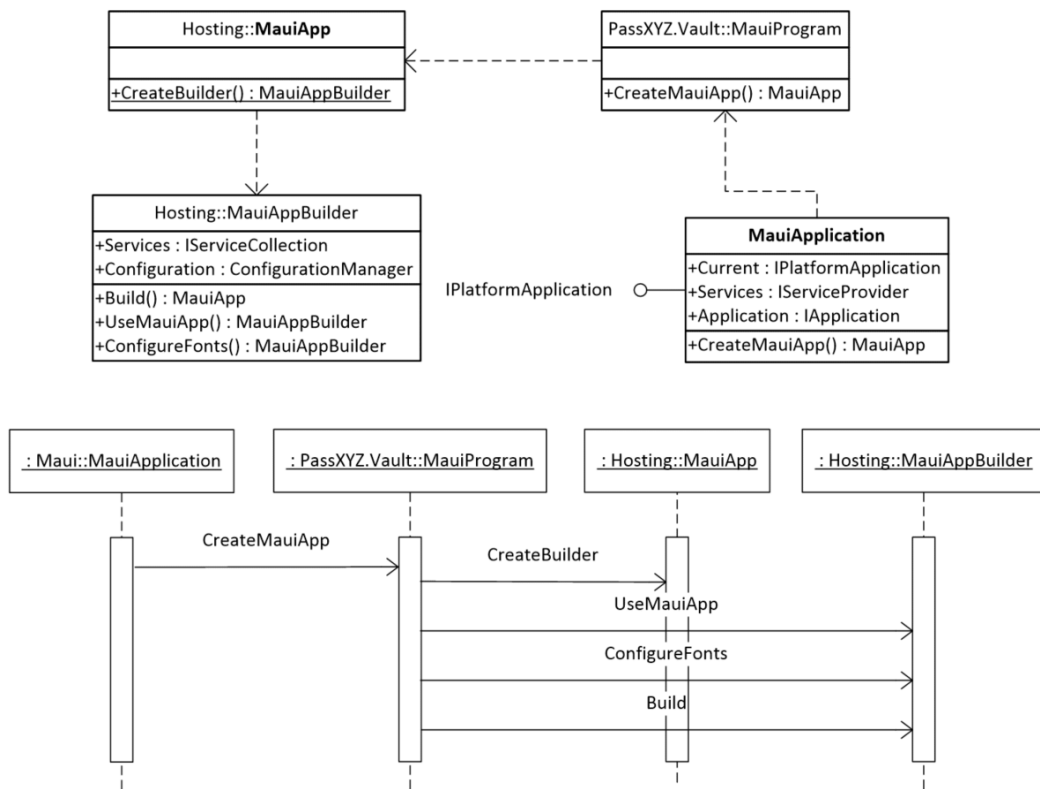


Figure 6.5: .NET MAUI DI configuration

At the top of *Figure 6.5*, we can see that there are four classes involved in the .NET MAUI application startup:

1. **Platform entry point**—The entry point of the .NET MAUI application is in platform-specific code. For the .NET MAUI project, it is in the `Platforms` folder. There are different classes defined for different platforms, as we can see in *Table 6.1*. In *Figure 6.5*, we use the `MauiApplication` Android version as an example:

Platform	Entry point class
Android	<code>MauiApplication</code>
iOS/macOS	<code>MauiUIApplicationDelegate</code>
Windows	<code>MauiWinUIApplication</code>

Table 6.1: Entry points in different platforms

All entry-point classes implement the `IPlatformApplication` interface, as we can see in the following code snippet:

```
public interface IPlatformApplication
{
    static IPlatformApplication? Current { get; set; }
    IServiceProvider Services { get; }
    IApplication Application { get; }
}
```

`IServiceProvider` is defined as part of this interface, so we can use it directly to resolve DI objects once the application is initialized.

2. **MauiProgram ❶**—As we can see in the code of the `MauiProgram` implementation shown next, each .NET MAUI app needs to define a static `MauiProgram` class and create a `CreateMauiApp()` method. The `CreateMauiApp()` method is invoked by the following override function, which is defined in all platform entry points. The return value is a `MauiApp` instance:

```
protected override MauiApp
    CreateMauiApp() => MauiProgram.CreateMauiApp();
```

3. **MauiApp ❷**—Inside `CreateMauiApp()`, it creates a `MauiAppBuilder` instance by calling `MauiApp.CreateBuilder()`.
4. **MauiAppBuilder ❸**—`MauiAppBuilder` includes a `Services` attribute of the `IServiceCollection` interface type. We can use it to configure DI for the .NET MAUI app.

From the previous analysis of the .NET MAUI app startup process, we can see that both `IServiceCollection` and `IServiceProvider` have been initialized during the startup process, so we can use them directly without further configuration.

We can refer to the following code snippet when we analyze the startup process. We registered two abstractions, `IDataStore` and `IUserService`, and a `LoginUser` class. They are all singleton objects:

```
public static class MauiProgram {
    public static MauiApp CreateMauiApp() {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts => {
                fonts.AddFont("fa-regular-400.ttf",
                    "FontAwesomeRegular");
            });
    }
}
```

```
        fonts.AddFont("fa-solid-900.ttf",  
            "FontAwesomeSolid");  
        fonts.AddFont("fa-brands-400.ttf",  
            "FontAwesomeBrands");  
        fonts.AddFont("OpenSans-Regular.ttf",  
            "OpenSansRegular");  
        fonts.AddFont("OpenSans-SemiBold.ttf",  
            "OpenSansSemiBold");  
    });  
    builder.Services.AddSingleton<IDataStore<Item>,  
        DataStore>();  
    builder.Services.AddSingleton<IUserService<User>,  
        UserService>();  
    builder.Services.AddSingleton<LoginUser, LoginUser>();  
    return builder.Build();  
}  
}
```

Once we have configured the interfaces and class, we can use them in our implementation. We can use the `IServiceProvider` interface to resolve objects. When we implement DI, there are three ways of injecting dependencies. We can use constructor injection, method injection, or property injection. Let's see how we can do it using the most common methods in the next two sections.

Constructor injection

In constructor injection, the dependencies required for the class are provided as arguments to the constructor. We can resolve dependencies using the constructor. This requires the registered dependency to have a parameterless constructor. In our code, the `LoginUser` class depends on the `IUserService` interface. The concrete class implementing `IUserService` is `UserService`, which defines a parameterless constructor. We can define the constructor of `LoginUser` as follows:

```
private IUserService<User> _userService;  
public LoginUser(IUserService<User> userService)  
{  
    _userService = userService ?? throw new  
        ArgumentNullException(nameof(userService));  
    _userService.CurrentUser = this;  
}
```

In the constructor of `LoginUser`, we list the dependency as a `userService` parameter. In this case, MS.DI will resolve `IUserService` as the configured `UserService` concrete class for us.

Property injection

There are many cases in which we won't be able to use constructor injection. In these cases, we can resolve the dependencies through `IServiceProvider`.

In the .NET MAUI application, the hosting environment creates an `IServiceProvider` interface for us, as we can see in *Figure 6.5*. We can use the `IPlatformApplication` interface defined in the platform-specific entry points to get the `IServiceProvider` interface, as shown in *Listing 6.1*:

Listing 6.1: ServiceHelper.cs (<https://epa.ms/ServiceHelper6-1>)

```
namespace PassXYZ.Vault.Services;

public static class ServiceHelper
{
    public static TService GetService<TService>()
        => Current.GetService<TService>(); ❷

    public static IServiceProvider Current => ❶
    #if WINDOWS10_0_17763_0_OR_GREATER
        MauiWinUIApplication.Current.Services;
    #elif ANDROID
        MauiApplication.Current.Services;
    #elif IOS || MACCATALYST
        MauiUIApplicationDelegate.Current.Services;
    #else
        null;
    #endif
}
```

❶ In the `ServiceHelper` class, we define a `Current` static variable to keep the reference of `IServiceProvider`, which is from the `IPlatformApplication` interface in platform entry points.

❷ A `GetService()` static method is defined that calls the `GetService()` method of `IServiceProvider`.

ServiceHelper

For the `ServiceHelper` implementation, I referred to the `MauiApp-DI` GitHub project. Thanks for James Montemagno's sample code in GitHub!

<https://github.com/jamesmontemagno/MauiApp-DI>

We can update our source code to replace `DependencyService` with DI with the help of `ServiceHelper`. In `BaseViewModel.cs`, we replaced `DependencyService` with DI, as shown next.

So, we replaced the following code:

```
public static IDataStore<Item> DataStore =>
    DependencyService.Get<IDataStore<Item>>();
```

This is what we replaced it with:

```
public static IDataStore<Item> DataStore =>
    ServiceHelper.GetService<IDataStore<Item>>();
```

You may feel that the preceding code of property injection doesn't look elegant compared to constructor injection. I haven't figured out a better way to do this in .NET MAUI. However, in the next part of this book, when we introduce the Blazor Hybrid app, we can resolve property injection using the C# attribute. To resolve the `IDataStore` interface in Blazor, we can do it in a much simpler way, as shown here:

```
[Inject]
public IDataStore<Item> DataStore { get; set; } = default!;
```

We can use the `[Inject]` C# attribute to resolve the dependency implicitly without calling the `GetService()` method of `ServiceHelper` explicitly.

When we move from `DependencyService` to DI, we will create another concrete class for the `IDataStore` interface. This class will handle the CRUD operations of the password database.

Connecting to the database

The password database is a local database in KeePass 2.x format. Inside the database, password data is stored as groups and entries. In the `KeePassLib` namespace, a `PwDatabase` class is defined to manage database operations. We can refer to the class diagram in *Figure 6.6* to understand the relationship between `PwDatabase`, `PwGroup`, and `PwEntry`:

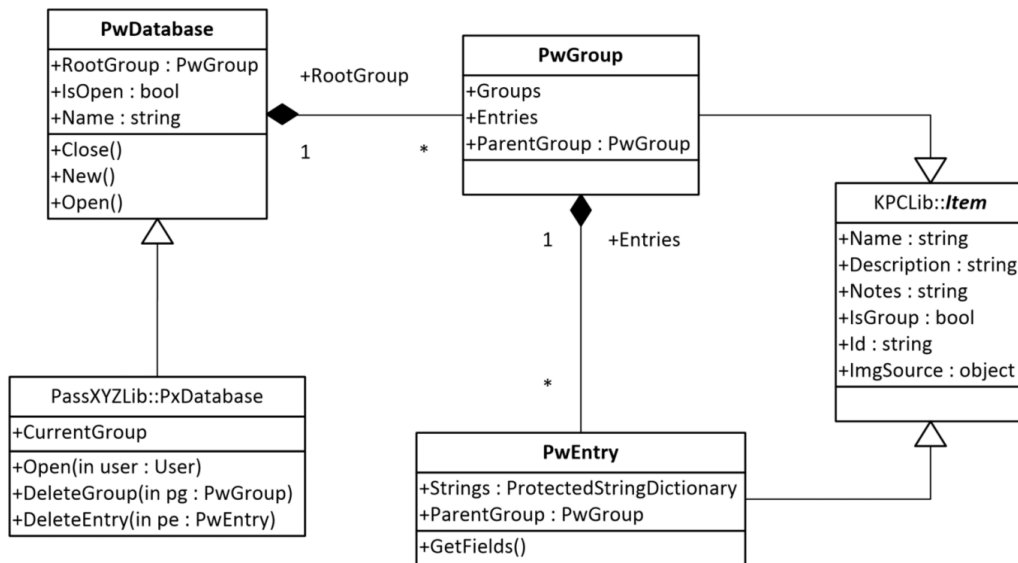


Figure 6.6: Class diagram of KeePass database

In **PwDatabase**, a **RootGroup** property of type **PwGroup** is defined. It contains all groups and entries stored in the database. We can navigate the data structure of the KeePass database from **RootGroup** to a particular entry. In **PwEntry**, a set of standard fields is defined, as shown in Figure 6.7:

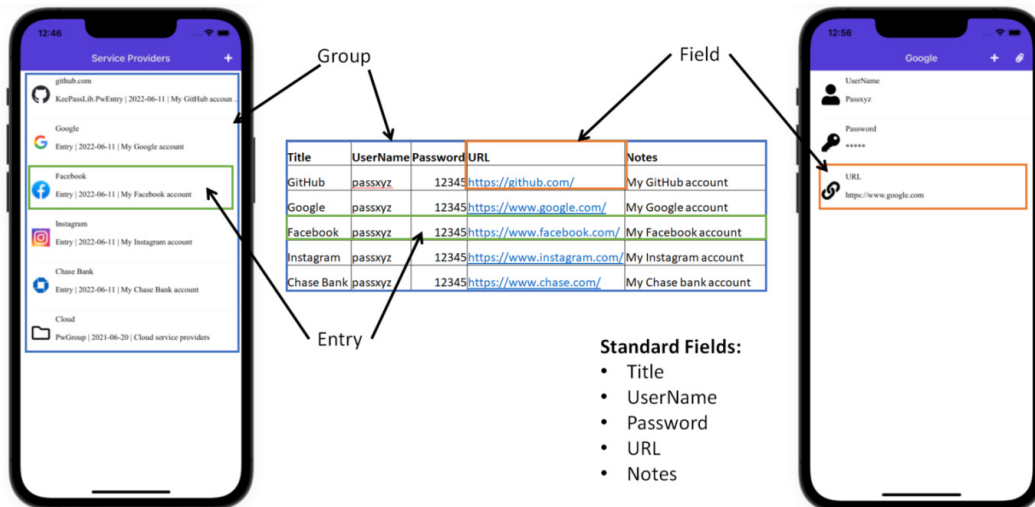


Figure 6.7: Group, entry, and field

If we have a list of entries that include only standard fields, this list looks like a table. In *Figure 6.7*, the current group includes five entries (**GitHub**, **Google**, **Facebook**, **Instagram**, and **Chase Bank**) and a sub-group (**Cloud**). On the left, there is a screenshot of `ItemsPage`, which shows the items in the current group. If the **Google** item was selected, it would be displayed as an entry in the screenshot on the right-hand side. Users may add additional fields to the entry, so the KeePass database is not a relational database—it is more like a key-value database. Each key-value pair is a field, such as a URL field.

To use `PwDatabase` in our app, a derived class, `PxDatabase`, is defined. `PxDatabase` added additional properties and methods such as `CurrentGroup`, `DeleteGroup()`, `DeleteEntry()`, and so on.

To access a database, we can open the database file and perform CRUD operations on it. Since we are building a cross-platform app, it is not convenient to handle the database file directly for the end users. In `PassXYZ.Vault`, the user concept is used instead of a data file. In `PassXYZLib`, a `User` class is defined to encapsulate the underlying file operations.

To access the database, we defined database initialization and CRUDL operations in the `IDataStore` and `IUserService` interfaces. The `DataStore` and `UserService` concrete classes are used to implement these two interfaces.

Initializing the database

The initialization of the database is part of the login process, so the following login method is defined in the `IUserService` interface:

```
Task<bool> LoginAsync(T user);
```

In the `UserService` class, `LoginAsync()` is defined as an async method, as we can see here:

```
public async Task<bool> LoginAsync(User user) {
    if (user == null) {
        Debug.Assert(false); throw new ArgumentNullException("user");
    }
    return await Task.Run(() => {
        if (string.IsNullOrEmpty(user.Password)) { return false; }
        db.Open(user);
        if (db.IsOpen) {
            db.CurrentGroup = db.RootGroup;
        }
        return db.IsOpen;
    });
}
```

In `LoginAsync()`, ❶ a separate task is used to handle the open operation of the database. The `Open()` ❷ method of `PxDatabase` is called, and an instance of the `User` class is passed to the `Open()` method as an argument.

Performing CRUD operations

The data operation of the KeePass database is similar to the CRUD operations in a relational database. Once we log in and connect to a database, we can access our password data. The first step is to retrieve a list of items. After login, the first list is retrieved from the root group. There is a read-only property, `RootGroup` ❶, which is defined in the `IDataStore` interface, as we can see in the following code snippet. Later, when the user navigates to a different group, the `CurrentGroup` ❷ property is used to keep the current location in the navigation:

Listing 6.2: `IDataStore.cs` (<https://epa.ms/IDataStore6-2>)

```
public interface IDataStore<T> {
    #region DS_misc
    T CurrentGroup { get; set; }           ❷
    string CurrentPath { get; }
    T RootGroup { get; }                  ❶
    bool IsOpen { get; }
    string GetStoreName();
    DateTime GetStoreModifiedTime();
    Task<bool> MergeAsync(string path);
    ObservableCollection<PxIcon> GetCustomIcons(...);
    Task<bool> DeleteCustomIconAsync(PxIcon icon);
    ImageSource GetBuiltInImage(PxIcon icon);
    #endregion

    DS_Item ...
}
```

Adding an item

The first operation in CRUD is a **create** or **add** operation. We can add an item that can be an entry or a group to the current group. The user interface for an add operation is a toolbar item in `ItemsPage`, as shown here:

```
<ContentPage.ToolbarItems>
    <ToolbarItem Text="{x:Static resources:Resources.
```

```
        action_id_add}" Command="{Binding AddItemCommand}" ">
    <ToolBarItem.IconImageSource>
        <FontImageSource FontFamily="FontAwesomeSolid"
            Glyph="{x:Static styles:FontAwesomeSolid.Plus}"
            Color="{DynamicResource SecondaryColor}"
            Size="16" />
    </ToolBarItem.IconImageSource>
</ToolBarItem>
</ContentPage.ToolbarItems>
```

We can see a toolbar item icon is shown in the top-right corner of ItemsPage in *Figure 6.8*:

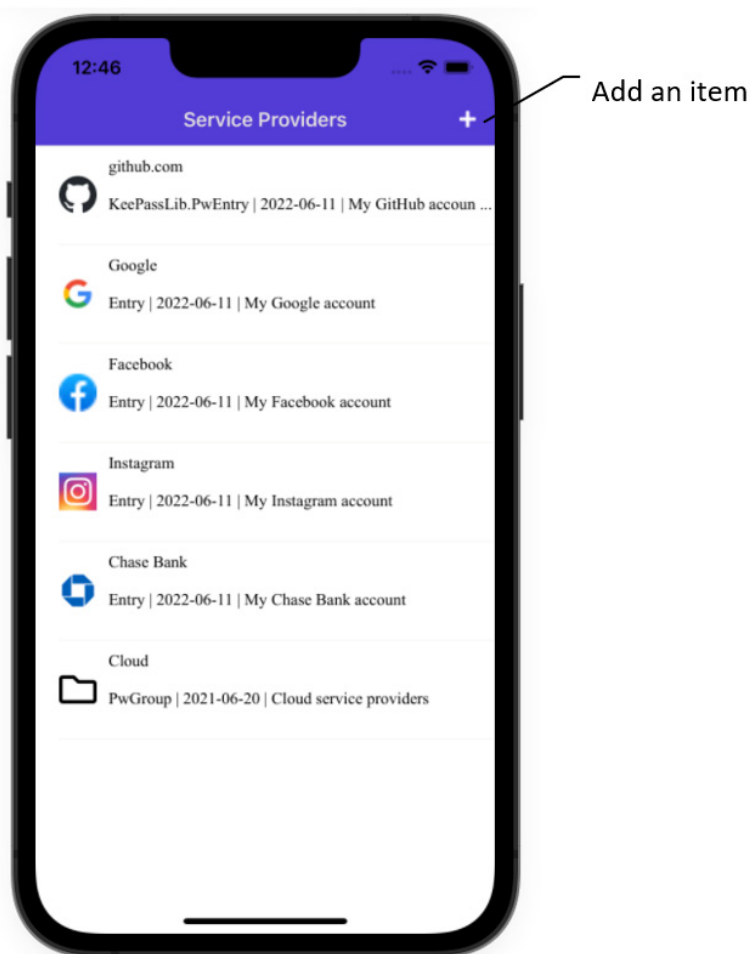


Figure 6.8: Adding an item

When the **Add** button is clicked, it invokes the `AddItemCommand` command defined in `ItemsViewModel` through data binding.

The `AddItemCommand` command invokes the following `OnAddItem()` method in the view model:

Listing 6.3: ItemsViewModel.cs (<https://epa.ms/ItemsViewModel6-3>)

```
private async void OnAddItem(object obj) {
    string[] templates = {
        Properties.Resources.item_subtype_group,
        Properties.Resources.item_subtype_entry,
        Properties.Resources.item_subtype_notes,
        Properties.Resources.item_subtype_pxentry
    };
    var template = await Shell.Current.DisplayActionSheet(
        Properties.Resources.pt_id_choosetemplate,
        Properties.Resources.action_id_cancel, null,
        templates); ❶

    ItemSubType type = ItemSubType.None;
    if (template == Properties.Resources.item_subtype_entry) {
        type = ItemSubType.Entry;
    } else if (template == Properties.Resources.item
        _subtype_pxentry) {
        type = ItemSubType.PxEntry;
    } else if (template == Properties.Resources.
        item_subtype_group) {
        type = ItemSubType.Group;
    } else if (template == Properties.Resources.item
        _subtype_notes) {
        type = ItemSubType.Notes;
    } else if (template == Properties.Resources.action
        _id_cancel) {
        type = ItemSubType.None;
    } else {
        type = ItemSubType.None;
    }
}
```

```

        if (type != ItemSubType.None) {
            var itemType = new Dictionary<string, object>
            {
                { "Type", type }
            };
            await Shell.Current.GoToAsync(nameof(NewItemPage),
                itemType);
        }
    }
}

```

❶ In this `OnAddItem()` function, `ActionSheet` is displayed to let the user choose an item type. The item type can be a group or an entry.

❷ Once we get the item type, we can build a dictionary with the item type and the name of the query parameter. We store this object of the dictionary in an `itemType` variable.

❸ This `itemType` variable can be passed to `NewItemPage` as a query parameter. In *Chapter 5, Introducing Shell and Navigation*, we learned how to pass a string value as a query parameter to a page in `Shell` navigation. Here, we can pass an object as a query parameter to a page after we wrap it in a dictionary.

To add a new item, the user interface is defined in `NewItemPage`, and the logic is processed in `NewItemViewModel`. Let's review the implementation of `NewItemViewModel` in *Listing 6.4*:

Listing 6.4: `NewItemViewModel.cs` (<https://epa.ms/NewItemViewModel6-4>)

```

using KPCLib;
using PassXYZLib;
namespace PassXYZ.Vault.ViewModels;

[QueryProperty(nameof(Type), nameof(Type))]
public class NewItemViewModel : BaseViewModel {
    private string text;
    private string description;
    private ItemSubType _type = ItemSubType.Group;

    public NewItemViewModel() {
        SaveCommand = new Command(OnSave, ValidateSave);
    }
}

```

```
CancelCommand = new Command(OnCancel);
this.PropertyChanged +=
    (_, __) => SaveCommand.ChangeCanExecute();
Title = "New Item";
}

private void SetTitle(ItemSubType type)...
private bool ValidateSave()...
public ItemSubType Type {                                ②
    get => _type;
    set {
        _ = SetProperty(ref _type, value);
        SetTitle(_type);
    }
}
public string Text...
public string Description...
public Command SaveCommand { get; }
public Command CancelCommand { get; }
private async void OnCancel() {
    await Shell.Current.GoToAsync(".."); }
private async void OnSave() {
    Item? newItem = DataStore.CreateNewItem(_type);      ③

    if (newItem != null) {
        newItem.Name = Text;
        newItem.Notes = Description;
        await DataStore.AddItemAsync(newItem);          ④
    }
    await Shell.Current.GoToAsync("..");
}
}
```

The design of `NewItemPage` is very simple. It includes two controls, `Entry` and `Editor` (used to edit the name and notes of an item). `Entry` is used to enter or edit a single line of text, and `Editor` is used to edit multiple lines of text. In the view model `NewItemViewModel` view model, we can see how to add a new item, as follows:

① The query parameter is defined with `QueryPropertyAttribute`. ② The `Type` property declared as `ItemSubType` is used to receive the query parameter. The received item type is stored in the `_type` backing variable. In `NewItemPage`, two toolbar items are defined, and the actions are bound to the `OnSave` and `OnCancel` methods in the view model.

Once the user enters a name and notes in the user interface and clicks the **Save** button, ③ a new item instance is created using the `CreateNewItem()` factory method, which is defined in the `IDataStore` interface. ④ After filling in the new item instance from the user input, we can add this new item to the database by invoking the `AddItemAsync()` method.

Now we've implemented the add operation, let's implement the rest of the data operations in the next section.

Editing or deleting an item

In CRUD operations, we don't need an existing item to perform a create operation, but we need to have an instance of the existing item to perform update and delete operations.

For a read operation, if the item is a group, we implement it by sending an `ItemId` query parameter to `ItemsPage` and finding the group in the setter of `ItemId` in the `ItemsViewModel` view model. If the item is an entry, we send an `ItemId` query parameter to `ItemDetailPage` and find the entry in the setter of `ItemId` in `ItemDetailViewModel`.

For update/edit and delete operations, we can use context actions to do it. With context actions, we can act on an item in `ListView`. The context actions look quite different on *iOS*, *Android*, and *Windows*, as we can see in *Figure 6.9*:

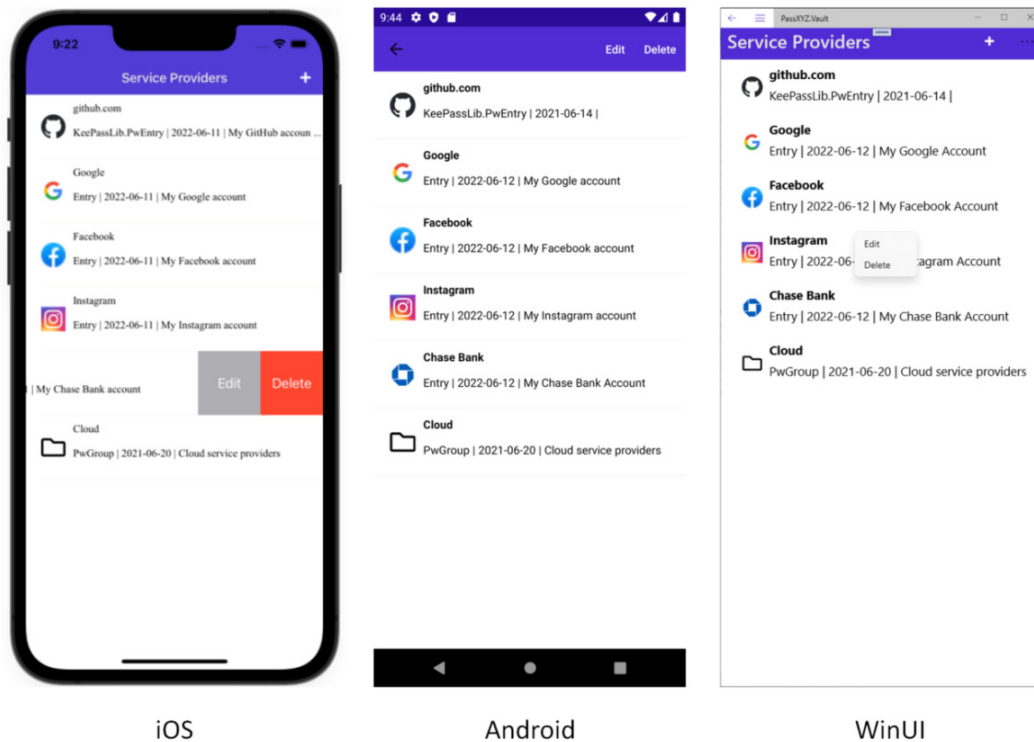


Figure 6.9: Context actions

On the iOS platform, you can take action on an item by sliding it to the left. On an Android system, you can long-press an item and the context actions menu is shown in the top-right corner of the screen. On Windows, you may be familiar with the right mouse click to see the context actions menu.

In our app, we implement a context actions menu in `ItemsPage`. In `ItemsPage`, we define context actions in `ViewCell`, as follows:

```
<ListView.ItemTemplate>
  <DataTemplate>
    <ViewCell>
      <ViewCell.ContextActions>
        <MenuItem Clicked="OnMenuEdit"                                ❶
          CommandParameter="{Binding .}"
          Text="{x:Static resources:Resources.
            action_id_edit}" />
        <MenuItem Clicked="OnMenuDeleteAsync"                          ❷

```

```

        CommandParameter="{Binding .}"
        Text="{x:Static resources:Resources.
            action_id_delete}"
        IsDestructive="True" />
    </ViewCell.ContextActions>

    <Grid Padding="10" x:DataType="model:Item" ...>
</ViewCell>
</DataTemplate>
</ListView.ItemTemplate>

```

We define two menu items for edit and delete context actions. Two event handlers, ❶ `OnMenuEdit` and ❷ `OnMenuDeleteAsync`, are assigned to the `Clicked` event. We can review the source code of the event handlers here:

```

private void OnMenuEdit(object sender, EventArgs e) {
    var mi = (MenuItem)sender;
    if (mi.CommandParameter is Item item) {
        _viewModel.Update(item);
    }
}
private async void OnMenuDeleteAsync(object sender,
    EventArgs e) {
    var mi = (MenuItem)sender;
    if (mi.CommandParameter is Item item) {
        await _viewModel.DeletedAsync(item);
    }
}

```

The `OnMenuEdit` and `OnMenuDeleteAsync` event handlers call the ❶ `Update()` and ❷ `DeletedAsync()` methods in the view model. Let's review the source code of these functions in the `ItemsViewModel` view model, as follows:

```

public async void Update(Item item) {
    if (item == null) { return; }
    await Shell.Current.Navigation.PushAsync(new
        FieldEditPage(async (string k, string v, bool
            isProtected) => {

```

```
        item.Name = k;
        item.Notes = v;
        await DataStore.UpdateItemAsync(item);
    }, item.Name, item.Notes, true));
}

public async Task DeletedAsync(Item item) {
    if (item == null) { return; }
    if (Items.Remove(item)) {
        _ = await DataStore.DeleteItemAsync(item.Id);
    }
    else { return; }
}
```

In `ItemsViewModel`, to edit or update an item, ❶ we use a `FieldEditPage` content page to perform the editing. When we invoke the constructor of `FieldEditPage`, an anonymous function is passed as a parameter. When the user completes the editing in `FieldEditPage`, this function will be invoked. In this function, ❷ we call the `UpdateItemAsync()` method of the `IDataStore` interface to update the item.

The delete operation is quite simple. We can just call the ❸ `DeleteItemAsync()` method of the `IDataStore` interface to remove the item.

After we implement CRUD operations, our app has most of the desired features of a password manager app. We can create a new database by signing up a new user. After we have a new database, we can log in to access our data. After we create entries and groups, we can also edit or delete them.

For a password manager app, there are more desired features, such as fingerprint scanning, one-time password, and so on. Most of these desired features are already included in the `PassXYZ.Vault` 1.x.x releases. I will continue migrating features from 1.x.x to the .NET MAUI (2.x.x) releases when the dependencies are available for .NET MAUI.

Device features

We can access device features using the class in the `Microsoft.Maui.Devices` namespace. The device features implementation originated from `Xamarin.Essentials` and then changed to `Maui.Essentials` in .NET MAUI preview versions. It finally became `Microsoft.Maui.Devices` in GA releases. Not all device features can be found in `Microsoft.Maui.Devices`, such as fingerprint scanning. To support fingerprint scanning in .NET MAUI, I need to wait for a library such as `Plugin.Fingerprint`, available in .NET MAUI, to enable it in `PassXYZ.Vault`.

Summary

In this chapter, we started with the introduction of design principles. After that, we introduced SOLID design principles and I shared lessons learned in the design of our app. One of the most important SOLID principles is **Dependency Inversion Principle (DIP)**. **Dependency Injection (DI)** is the technique to apply DIP in the actual implementation. In our app, we use the .NET MAUI built-in DI service to decouple dependencies so that we can separate the implementation of the service from the interface.

With all the knowledge that we gathered about .NET MAUI, we completed our app implementation by replacing `MockDataStore` with the actual implementation. We implemented CRUD operations on top of this new `IDataStore` service. We have a fully functional password manager app now.

With the current version of the password manager app, we conclude *Part 1* of this book.

In *Part 2* of the book, we will explore the Blazor Hybrid app in .NET MAUI. This is a new capability that does not exist in `Xamarin.Forms`. With Blazor support, we can bring some modern frontend development techniques to .NET MAUI development.

Further reading

- **Autofac** is an **inversion of control (IoC)** container for .NET Core, ASP.NET Core, .NET 4.5.1+, and more:

<https://autofac.org/>

- **Simple Injector** is a DI container that can support .NET 4.5 and .NET Standard:

<https://simpleinjector.org/>

Part 2:

Implementing .NET

MAUI Blazor

In the second part of this book, we will learn how to build a .NET MAUI Blazor Hybrid app. Blazor is a **single-page app (SPA)** framework that uses Razor components as building blocks. You might have heard of other SPA frameworks, such as React, Angular, Vue, and so on. Most SPA frameworks use JavaScript, but Blazor uses C# instead of JavaScript. Blazor can also be used to build the so-called Blazor Hybrid application. In a Blazor Hybrid app, Razor components run natively on the device using an embedded Web View control so that the Blazor Hybrid app can access device features in the same way as a native app. We will re-build our app as a Blazor Hybrid app in *Part 2*. There are four chapters in *Part 2*.

This section comprises the following chapters:

- *Chapter 7, Introducing Blazor Hybrid App Development*
- *Chapter 8, Understanding the Blazor Layout and Routing*
- *Chapter 9, Implementing Blazor Components*
- *Chapter 10, Advanced Topics in Creating Razor Components*

Introducing Blazor Hybrid App Development

In .NET MAUI, there is another way to build the user interface, which is using Blazor. Blazor can be used as a **single-page application (SPA)** framework to develop web applications. It can also be used to develop .NET MAUI apps in the form of a Blazor Hybrid app. The building block of Blazor is the Razor component. Using Blazor and Blazor Hybrid, we can reuse Razor components between native apps and web apps. Compared to the XAML UI, the Blazor UI can achieve higher reusability, which includes both native apps and web apps. In this chapter, we will introduce what Blazor is and explain its usage in different scenarios. We will also introduce the Razor component and how to develop a Blazor Hybrid app using Razor components.

We will cover the following topics in this chapter:

- What is Blazor?
- How to create a .NET MAUI Blazor project
- How to create a new Razor component

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code for this chapter is available in the following GitHub repository: <https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter07>.

The source code can be downloaded using the following `git` command:

```
git clone -b chapter07 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

What is Blazor?

Blazor is a framework for building web applications using **HTML**, **CSS**, and **C#**. To build web applications using Blazor in ASP.NET Core, there are two options to choose from, which are Blazor Server and **Blazor WebAssembly (Wasm)**. With .NET MAUI, Blazor can also be used to build native applications, and this is the third form – a Blazor Hybrid app.

In web application development, the tasks usually include creating a frontend user interface and backend service. Backend services are accessed through the RESTful API or **remote procedure calls (RPCs)**. The components of user interfaces consist of HTML, CSS, and JavaScript. They are loaded in a browser and displayed as web pages. We can render components related to the user interaction on the server in the ASP.NET Core architecture. This hosting model is Blazor Server. We can also choose to execute most of the user interface components in the browser; this hosting model is called Blazor Wasm. In some applications, we have requirements for the application to access device-specific features, such as sensors or cameras. In that case, we usually need to develop native applications to meet such requirements. With Blazor, we have another choice, which is the Blazor Hybrid app. First, let's talk about Blazor Server.

Learning about Blazor Server

In traditional web application development, user interaction logic is performed on the server side. In the MVC design pattern, processing user interaction is part of the application architecture. If there is a user interaction in the browser, it is sent back to the server to be processed. In response to the user's request, the entire page may be reloaded. To improve the performance, Blazor Server uses a design similar to an SPA framework. To respond to the user request, Blazor Server processes the request, and it only sends the **Document Object Model (DOM)** changes related to the user action to the browser. In Blazor Server, as we can see in *Figure 7.1*, the processing logic is the same as an SPA, with the difference being that Razor components are rendered on the server instead of the browser. **SignalR**, an open source library that can be used for real-time communication between the web client and the server, is used as the connection between the server and the browser:

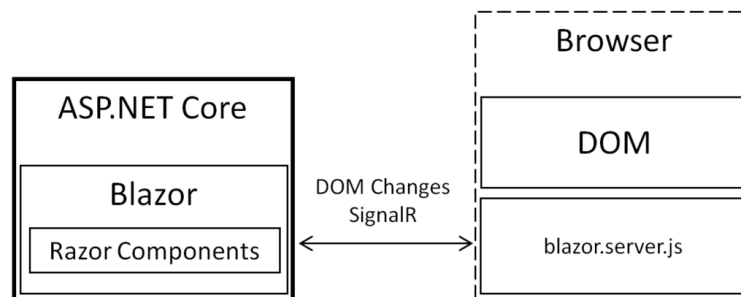


Figure 7.1: Blazor Server

Razor components versus Blazor components

People may get confused between Blazor and Razor. Razor was introduced as a template engine of ASP.NET in 2010. Razor syntax is a markup syntax in which developers can embed C# code into an HTML page. Blazor is an SPA framework that uses Razor syntax as the programming language. It was introduced around 2018. Blazor is a component-based framework, and a Blazor app consists of Razor components. In other words, Blazor is a hosting model for Razor components. Blazor components and Razor components are widely used interchangeably, but the correct terminology is Razor component.

A Razor component resides in a file with the `.razor` extension, and it is compiled as a .NET class at runtime. This `.razor` file can also be split into two files with `.razor` and `.razor.cs` extensions. The idea is quite similar to XAML and code-behind, which we learned about in *Part 1* of this book. We will learn how to build Razor components using Razor syntax in this chapter. Next, we will talk about Blazor Wasm.

Understanding Blazor Wasm

Blazor **Wasm** is a hosting model that renders Razor components in a web browser. Razor components loaded in the browser are compiled in Wasm using the .NET runtime, as we can see in *Figure 7.2*:

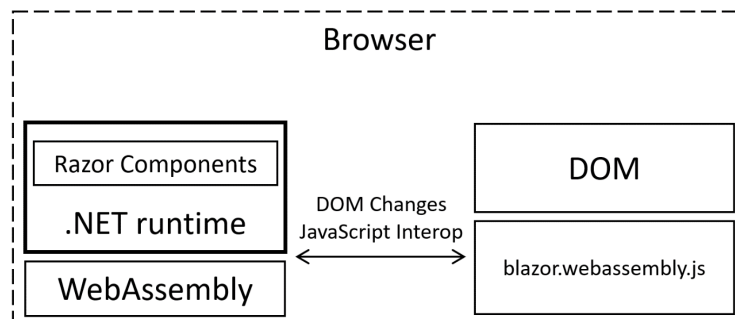


Figure 7.2: Blazor Wasm

In the browser, the startup page will load the .NET environment and Razor components. Razor components are compiled to Wasm using a .NET Intermediate Language (IL) interpreter at runtime to handle DOM changes. This is so-called **just-in-time (JIT)** compilation. Using JIT, the compilation happens at runtime, so the performance is slower than **ahead-of-time (AOT)** compilation. Using AOT, the compilation is done at development time, so the runtime performance can be improved. Blazor Wasm supports **mixed-mode** AOT compilation in .NET 7, which means part of .NET code can be compiled in Wasm at the development stage to gain a significant runtime performance improvement. Mixed-mode AOT means that only part of the IL code can be compiled into Wasm due to technical reasons.

Wasm

Wasm is a binary instruction format for a stack-based virtual machine. Wasm is supported by most modern web browsers. With Wasm, we can use many programming languages to develop client-side components.

As an SPA framework, we can compare Blazor to other JavaScript-based SPA frameworks. There are many JavaScript SPA frameworks, such as React, Angular, and Vue. In *Table 7.1*, we have compared Blazor with React. Of course, we can use other JavaScript frameworks for the comparison as well. The reason that I chose React is that React Native can be used to develop Hybrid apps, which has some similarities to .NET MAUI Blazor, which we will discuss in the next section:

Feature	React	Blazor Wasm	Blazor Server
Language	JavaScript/JSX/TypeScript	C#	C#
Runtime	JavaScript engine	Wasm	ASP.NET Core
Progressive Web App (PWA) Support	Yes	Yes	No
Hosting	Flexible to choose	Flexible to choose	ASP.NET Core
Static Site Hosting	Yes	Yes	No
Offloads Processing to Clients	Yes	Yes	No
Performance	Lightweight with great performance	There is a heavier first-time load due to the extra download time of .NET runtimes	Similar performance to the JavaScript framework

Table 7.1: Comparison of Blazor and React

Both JavaScript and Wasm are built-in features of modern browsers. The SPA frameworks using either JavaScript or Wasm do not have any extra dependencies to run in a browser.

Blazor Wasm supports JavaScript Interop, so JavaScript components can be used by Blazor as well. Many JavaScript libraries have been ported to Blazor.

Both Blazor and React support PWA development, which makes SPAs accessible in offline mode. The first-time loading performance of Blazor is slightly heavier than React due to the extra download time of .NET runtimes.

Exploring Blazor Hybrid

We can use Blazor as the user interface layer in desktop or mobile native frameworks, known as Blazor Hybrid apps. In a Blazor Hybrid app, Razor components are rendered natively on the device using an embedded WebView control. Wasm isn't involved, so the app has the same capability as a native app.

As we can see in *Figure 7.3*, in a Hybrid app, we can use the `BlazorWebView` control to build and run Razor components inside an embedded WebView. The `BlazorWebView` control is available in .NET MAUI and Windows desktop environments. We can build Blazor Hybrid applications in .NET MAUI, WPF, or Windows Forms:

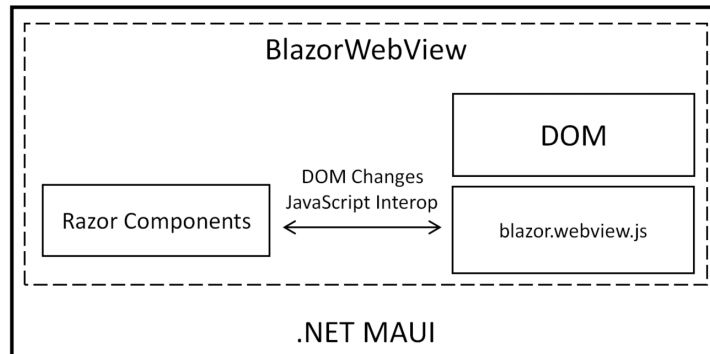


Figure 7.3: BlazorWebView

Blazor Server, Blazor Wasm, and Blazor Hybrid run in different runtime environments, so they have different capabilities. We can create different project types using either the command line or Visual Studio.

To list the installed project templates, we can use the following command:

```
dotnet new --list
```

These templates matched your input:

Template Name	Short Name	Language
Blazor Server App	blazorserver	[C#]
Blazor WebAssembly App	blazorwasm	[C#]
Razor Class Library	razorclasslib	[C#]
.NET MAUI App	maui	[C#]
.NET MAUI Blazor App	maui-blazor	[C#]
.NET MAUI Class Library	maulib	[C#]
Class Library	classlib	[C#], F#, VB

In the preceding list, we filtered out irrelevant project types. To understand the different project types better, we can review the summary depicted in *Table 7.2*:

Template Name/Short Name	SDK	Target Framework
Blazor Wasm App (blazorwasm)	Microsoft.NET.Sdk.BlazorWasm	net6.0
Blazor Server App (blazorserver)	Microsoft.NET.Sdk.Web	net6.0
.NET MAUI App (maui)	Microsoft.NET.Sdk	net6.0-android net6.0-ios net6.0-maccatalyst net6.0-windows10.0.19041.0
.NET MAUI Blazor App (maui-blazor)	Microsoft.NET.Sdk.Razor	net6.0-android net6.0-ios net6.0-maccatalyst net6.0-windows10.0.19041.0
.NET MAUI Class Library (mauilib)	Microsoft.NET.Sdk	net6.0-android net6.0-ios net6.0-maccatalyst net6.0-windows10.0.19041.0
Razor Class Library (razorclasslib)	Microsoft.NET.Sdk.Razor	net6.0
Class Library (classlib)	Microsoft.NET.Sdk	net6.0

Table 7.2: .NET MAUI and Blazor-related project types

There are seven project types listed in *Table 7.2*. They can be divided into two groups – Blazor app and .NET MAUI app. Let's take a look at them.

Blazor apps

For both Blazor Server and Blazor Wasm, the target framework is net6.0, but they use different SDKs. A Blazor Server app can take full advantage of the server's capability with **Microsoft.NET.Sdk.Web**, while Blazor Wasm can only access a limited set of .NET API using **Microsoft.NET.Sdk.BlazorWasm**.

To share Razor components between Blazor Server and Blazor Wasm, we can use a Razor Class Library, which uses **Microsoft.NET.Sdk.Razor**. The standard .NET class library, which can be shared by all .NET 6.0 apps, uses **Microsoft.NET.Sdk**.

.NET MAUI apps

For a .NET MAUI App, we can build XAML-based .NET MAUI apps using **Microsoft.NET.Sdk**; .NET MAUI Blazor apps use **Microsoft.NET.Sdk.Razor**. Both project types target the same set of target frameworks.

To share components, the standard .NET class library can be used. If .NET MAUI features have to be used in the shared components, the .NET MAUI class library can be used. For example, `PassXYZLib` is a .NET MAUI class library. Both the .NET class library and the .NET MAUI class library use the same **Microsoft.NET.Sdk**, but they target different frameworks.

The Razor components in the .NET MAUI Blazor app have full access to the native capabilities of the device through the underlying platform SDK. The relationship between Blazor and Blazor Hybrid is similar to React and React Native. In a .NET MAUI Blazor app and React Native, both may have to implement some platform-specific code. We can use *Table 7.3* to compare how to access platform-specific APIs in these two frameworks.

Platform-specific implementation in .NET MAUI Blazor and React Native

In .NET MAUI, a full .NET implementation is available on the target operating system. For example, we have all .NET 6.0 APIs and almost a full .NET version of the Android APIs in `net6.0-android`. To implement a platform-specific feature in Android, we don't have to use Android Studio and Java/Kotlin. We can implement it in the .NET environment entirely. The recommended solution is to combine multi-targeting with partial classes and partial methods to invoke platform code from cross-platform code. This approach can improve the reusability of code and limit the platform-specific code to the minimum scope.

In React Native, native modules are used to implement platform-specific code. To implement native modules, platform-specific tools have to be used. For example, **Android Studio** is used to implement Android native modules and **Xcode** is used for iOS ones:

Platforms	.NET MAUI Blazor	React Native
Android	net6.0-android31.0/C#	Android Studio/Java or Kotlin
iOS	net6.0-ios15.2/C#	Xcode/Object-C or Swift
macOS	net6.0-maccatalyst15.2/C#	N/A
Windows	net6.0-windows10.0.19041/C#	N/A

Table 7.3: Comparison between .NET MAUI Blazor and React Native

There are pros and cons to both approaches. In the .NET MAUI implementation, both .NET- and platform-specific APIs are provided as .NET APIs in C#. We can maximize the reuse of the code to expose platform-specific implementations as a cross-platform API. However, the effort to maintain the target frameworks is huge.

In React Native, there is no standard platform-specific implementation available. The developers need to develop them on their own when they need to access platform-specific features. It looks flexible, but it is difficult to reuse code in the framework. Different developers may invent different libraries to access the same set of platform features. Native module developers must acquire different skills to develop cross-platform apps.

If we create a cross-platform solution using Blazor, it is possible to develop a solution that includes web, mobile, desktop, and backend services using one architecture and programming language. The shared components can be reused in all target applications.

The paradigm of React Native is to *learn once, write anywhere*. React and React Native both use the JavaScript language and similar frameworks, but they cannot reuse components as we can do in Blazor. React Native uses a native user interface, which is similar to the XAML-based .NET MAUI app, while Blazor Hybrid apps use HTML-based web user interfaces.

To summarize, Blazor Hybrid and React Native have quite different design goals, so their features and capabilities are different. You must understand your requirements first before choosing a framework.

Creating a new .NET MAUI Blazor project

To learn how to develop a Blazor Hybrid app, we must upgrade our `PassXYZ.Vault` project to support the Blazor-based UI. We don't have to do this from scratch – we can convert our current project so that it supports the Blazor UI. In this way, we can build both a XAML-based app and a Hybrid app using the same project. Before we add the Blazor UI to our app, we can create a new .NET MAUI Blazor project with the same app name first so that we can refer to this new project to convert our project into a .NET MAUI Blazor project.

We can create a new .NET MAUI Blazor project from the command line or Visual Studio. We will demonstrate both in this section.

Generating a .NET MAUI Blazor project with the dotnet command line

Let's create a new project using the .NET command line first. This can be done on either Windows or macOS.

We can create a new project using the shortname **maui-blazor** listed in *Table 7.2*:

```
dotnet new maui-blazor -o PassXYZ.Vault

Welcome to .NET 6.0!
-----
SDK Version: 6.0.402

Telemetry
-----
The .NET tools collect usage data in order to help us improve
your experience. It is collected by Microsoft and shared with
the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or
'true' using your favorite shell.

Read more about .NET CLI Tools telemetry: https://aka.ms/
dotnet-cli-telemetry
-----
Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate run 'dotnet dev-certs https --trust'
(Windows and macOS only).
Learn about HTTPS: https://aka.ms/dotnet-https
-----
Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/
dotnet/core
```

```
Use 'dotnet --help' to see available commands or visit:
https://aka.ms/dotnet-cli

-----
-----

The template ".NET MAUI Blazor App" was created successfully.
```

In the preceding command, we specified the short name of `maui-blazor` to choose the project template and we used `PassXYZ.Vault` as the project name. Once we have created the project, we can build and run it:

```
C:\> dotnet build -t:Run -f net6.0-android
MSBuild version 17.3.2+561848881 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  PassXYZ.Vault -> C:\PassXYZ.Vault\Hybrid\bin\Debug\net6.0-android\PassXYZ.Vault.dll
Build succeeded.
    0 Warning(s)
    0 Error(s)
Time Elapsed 00:01:43.79
```

If you are using a later version of Visual Studio, you may specify a different framework, such as `net7.0-android`.

In the build command, we specify `net6.0-android` as the target framework to test our new app. We can refer to *Figure 7.5* to see a screenshot of this new app and project structure. With that, we have created the new project using a command line. Now, let's learn how to do the same using Visual Studio on Windows. If you are using Visual Studio on a Mac, the process is very similar.

Creating a .NET MAUI Blazor project using Visual Studio on Windows

To create a .NET MAUI Blazor project using Visual Studio, we can start Visual Studio and select **Create a new project**, and then type **MAUI** in the search box. As we can see in *Figure 7.4*, we can select **.NET MAUI Blazor App** from the list of project templates:

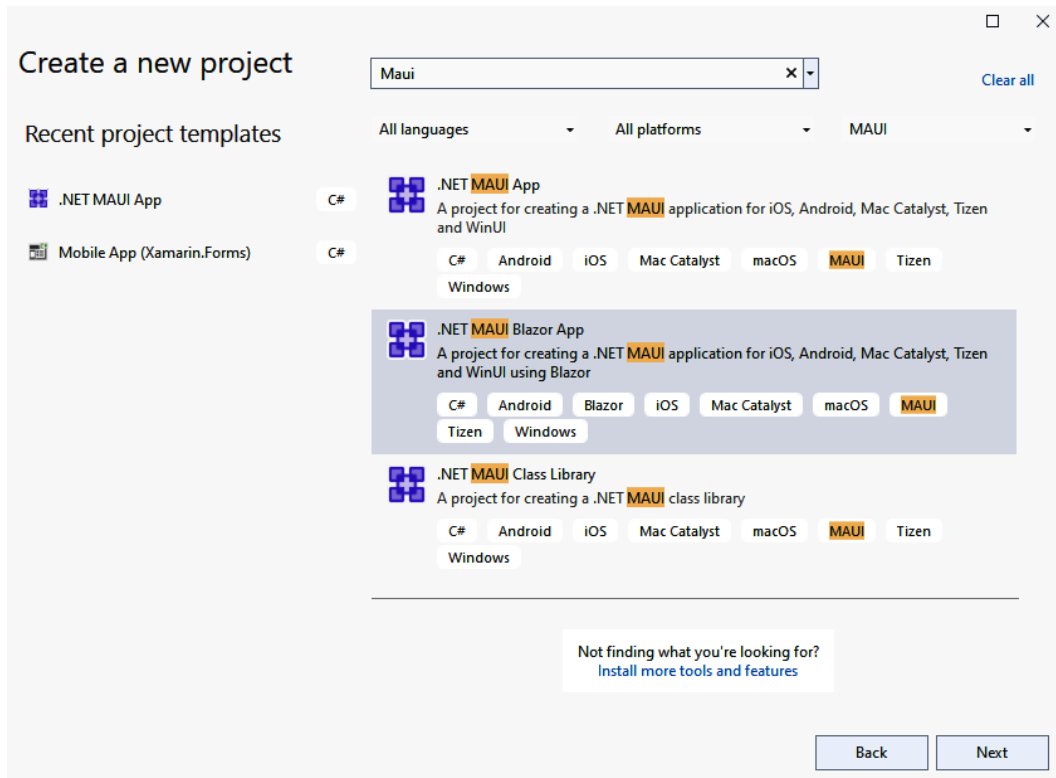


Figure 7.4: Creating a new .NET MAUI Blazor project

After following the wizard to create the project, we can select `net6.0-android` as the target framework to build and run it. To save space, we will use the Android platform as an example here; you can select other target frameworks to try and test if you like.

Running the new project

To run the project, we can press `F5` or `Ctrl + F5` in Visual Studio, or we can execute the `dotnet` command from the command line. We can see screenshots of this and the project structure in *Figure 7.5*:

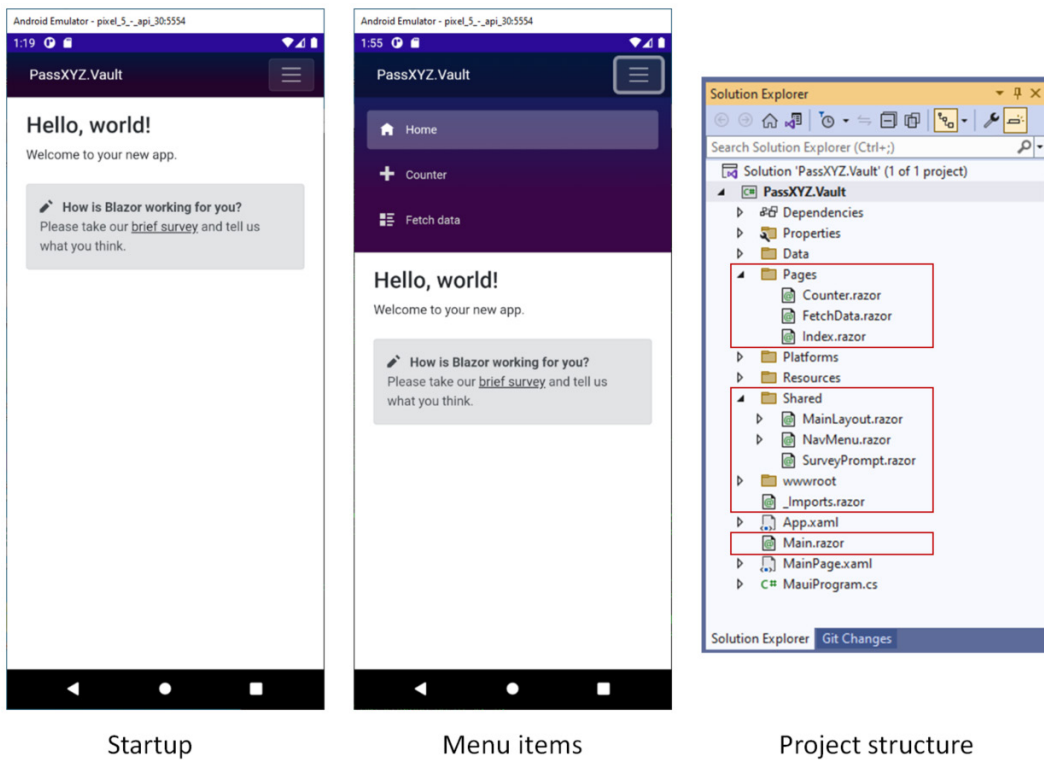


Figure 7.5: Screenshots and project structure

The user interface of the app created from the template looks like an SPA with a navigation menu at the top on Android. If we run it on Windows with a bigger screen, the navigation menu will be shown side by side on the left of the screen. The project structure is similar to a standard .NET MAUI app with the following differences:

- `wwwroot/`: This folder is the root of static files for web pages
- `Pages/`: This folder contains Razor pages in the app
- `Shared/`: This folder contains Razor components that can be shared
- `Main.razor`: This is the main page of the Blazor app
- `_Imports.razor`: This is a helper to import Razor components at the folder or project level

To understand the difference between the .NET MAUI app and the .NET MAUI Blazor app, we must review the startup code.

The startup code of the .NET MAUI Blazor app

All .NET MAUI apps include a file called `MauiProgram.cs` for their startup and configuration. Let's review the startup code of the .NET MAUI Blazor app:

```
namespace PassXYZ.Vault;

public static class MauiProgram {
    public static MauiApp CreateMauiApp() {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>()
            .ConfigureFonts(fonts => {
                fonts.AddFont("OpenSans-Regular.ttf",
                    "OpenSansRegular");
            });
        builder.Services.AddMauiBlazorWebView();           ❶
#if DEBUG
        builder.Services.AddBlazorWebViewDeveloperTools(); ❷
#endif
        builder.Services.AddSingleton<WeatherForecastService>();
        return builder.Build();
    }
}
```

In the .NET MAUI Blazor version, we can see that the following Blazor configurations are added:

- ❶ `BlazorWebView` is added by calling `AddMauiBlazorWebView()`
- ❷ Developer tools are added by calling `AddBlazorWebViewDeveloperTools()` for debugging

The rest of the startup process is the same as that for a XAML-based .NET MAUI app. In `App.xaml.cs`, the inherited `MainPage` property of the `App` class is set to `MainPage.xaml`:

```
namespace PassXYZ.Vault;

public partial class App : Application {
    public App() {
        InitializeComponent();
        MainPage = new MainPage();
    }
}
```

The major difference between XAML-based apps and Blazor Hybrid apps is in `MainPage.xaml`. Let's review `MainPage.xaml`:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com
  /dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:PassXYZ.Vault"
  x:Class="PassXYZ.Vault.MainPage"
  BackgroundColor="{DynamicResource PageBackgroundColor}">

  <BlazorWebView HostPage="wwwroot/index.html">           ①
    <BlazorWebView.RootComponents>                          ②
      <RootComponent Selector③="#app" ComponentType④=
        "{x:Type local:Main}" />
    </BlazorWebView.RootComponents>
  </BlazorWebView>
</ContentPage>
```

There is only one UI element, called `BlazorWebView`, defined in `MainPage.xaml`. In `BlazorWebView`, we can use the `HostPage` property and the `RootComponent` nested component to customize `BlazorWebView`.

We can treat `BlazorWebView` as a browser. The user interface of a browser is loaded from an HTML file. The `HostPage` property ① is used to specify the static HTML page to be loaded in the web view control. In our case, it is `wwwroot/index.html`; we will review it in *Listing 7.1*.

In this static HTML file, we need to specify where the Razor component should be placed, and which Razor component should be the root component. We can specify both using the `RootComponent` nested component ②.

In the previous chapter, we learned that a tag of XAML maps to a C# class eventually. Here, both `BlazorWebView` and `RootComponent` are C# classes as well.

In `RootComponent`, we use the `Selector` property ③ to define a CSS selector that specifies where the root Razor component in our app should be placed. In our case, it is the `#app` CSS selector defined in `index.html`. The `ComponentType` property ④ defines the type of the root component. In our case, it is `Main`.

Finally, let's review the HTML file (`index.html`) that we mentioned previously:

Listing 7.1: `index.html` (<https://epa.ms/index7-1>)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0, maximum-scale=1.0, user-
      scalable=no, viewport-fit=cover" />
  <title>PassXYZ.Vault</title>
  <base href="/" />
  <link rel="stylesheet" href="css/bootstrap/
    bootstrap.min.css" />
  <link href="css/app.css" rel="stylesheet" />
  <link href="PassXYZ.Vault.styles.css" rel="stylesheet" />
</head>
<body>
  <div class="status-bar-safe-area"></div>
  <div id="app">Loading...</div>
  <div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">X</a>
  </div>
  <script src="_framework/blazor.webview.js"
    autostart="false">
  </script>
</body>
</html>
```

We can see that `index.html` is a simple HTML file:

- ❶ It uses the CSS stylesheet from the Bootstrap framework.
- ❷ The id selector is defined as "app", which we pass to the `Selector` attribute of `RootComponent` in `MainPage.xaml`.

❸ A JavaScript file called `blazor.webview.js` is loaded at the end of `index.html`. It initializes the runtime environment of `BlazorWebView`.

With that, we have an overview of the .NET MAUI Blazor app. In the next section, we will replace the XAML-based user interface with a Blazor user interface.

Migrating to a .NET MAUI Blazor app

Instead of doing everything from scratch, we can use both the XAML and Blazor UIs in our app by changing the project configuration referring to the project that we created in the previous section. We will use a mixed XAML and Blazor user interface in one app for a while until we move everything to Blazor.

To convert our app into a .NET MAUI Blazor app, we need to make the following changes:

1. Change the SDK from `Microsoft.NET.Sdk` to `Microsoft.NET.Sdk.Razor` in the project file since the .NET MAUI Blazor app uses a different SDK.

In the `PassXYZ.Vault.csproj` project file, we have the following line:

```
<Project Sdk="Microsoft.NET.Sdk">
```

We need to replace it with the following line:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

2. Copy the following folders from the new project that we just created into our app:

- `wwwroot`
- `Shared`

3. Copy the following files from the new project into our app:

- `_Imports.razor`
- `MainPage.xaml`
- `MainPage.xaml.cs`
- `Main.razor`

4. Change `MauiProgram.cs` by adding the following code:

```
builder.Services.AddMauiBlazorWebView();  
#if DEBUG  
builder.Services.AddBlazorWebViewDeveloperTools();  
#endif
```

To review the commit history of these changes, go to <https://epa.ms/Blazor7-1>.

With these changes, we have made all the modifications we need to the configuration and are ready to move on to the next step. However, before we start to work on these changes, let's get familiar with the basic Razor syntax.

Understanding Razor syntax

Blazor apps consist of Razor components. As we learned in *Chapter 3, User Interface Design with XAML*, XAML is a type of language derived from the XML language. XAML-based UI elements consist of XAML pages and code-behind C# files. Razor components look very similar to this pattern. The difference is that Razor uses HTML as its markup language and C# code can be embedded into HTML directly. Optionally, we can also choose to separate C# code in a code-behind file to separate the UI and logic.

Code blocks in Razor

If we add a new Razor component to the project, it will look like this:

```
<h3>Hello World!</h3>
@code {
    // Put your C# code here
}
```

In the preceding example, we can design our page just like an HTML page and put programming logic into a code block. Razor pages or Razor components will be generated as C# classes. The filename is used as the class name. They can be used as HTML tags on another Razor page.

Implicit Razor expressions

In Razor syntax, we can transit from HTML to C# using the @ symbol. These are called implicit Razor expressions. For example, we can use the following implicit expression to set the text of the `label` tag with the `currentUser.Username` C# variable:

```
<label>@currentUser.Username</label>
```

There must not be any spaces between implicit expressions. We cannot use C# generics in implicit expressions since the characters inside the brackets (<>) are interpreted as an HTML tag.

Explicit Razor expressions

To resolve the issues of implicit expressions (such as white space or using generics), we can use explicit Razor expressions. Explicit Razor expressions consist of an @ symbol with parentheses. We can call a generic method like so:

```
<p>@(GenericMethod<int>())</p>
```

When we want to concatenate text with an expression, we also need to use explicit expressions, like so:

```
<p>@(currentUser.FirstName) _@(currentUser.LastName)</p>
```

We can use explicit Razor expressions in more complicated cases, such as passing a lambda expression to an event handler. Let's review another case of using an explicit Razor expression when we embed HTML inside C# code.

Expression encoding

Sometimes, we may want to embed HTML as a string inside C# code, but the result may not be what we expect.

Let's say we write the following C# expression:

```
@("<span>Hello World!</span>")
```

The result will look like this after rendering:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

To preserve the HTML string, we need to use the MarkupString keyword, like this:

```
@((MarkupString)"<span>Hello World</span>")
```

The result of the preceding C# expression is as follows:

```
<span>Hello World!</span>
```

This is the output that we want. We will learn more about explicit Razor expressions when we create Razor components.

Directives

Besides HTML code and C# code blocks, there are a set of reserved keywords to be used as Razor directives. Razor directives are represented by implicit expressions with reserved keywords after the `@` symbol. We saw the code block as `@code` in the previous section. Here, `@code` is a directive with a reserved keyword of `code`. The following are the directives that we can use in this book:

- `@attribute`: This is used to add the given attribute to the class
- `@code`: This is used to define a code block
- `@implements`: This is used to implement an interface for the generated class
- `@inherits`: This is used to specify the parent class for the generated class
- `@inject`: This is used to inject a service using dependency injection
- `@layout`: This is used to specify a layout for routable Razor components
- `@namespace`: This is used to define the namespace for the generated class
- `@page`: This is used to define a route for the page
- `@using`: This is similar to the `using` keyword in C#, which imports a namespace

Directive attributes

In a Razor page, HTML tags can be classes and attributes can be members of the class. Let's review the following example:

```
<input type="text" @bind="currentUser.Username">
```

Here, `input` is an HTML tag, which is a class. The `type` attribute is the property of the `input` tag, which is assigned to the `"text"` string. You may have noticed that another attribute called `@bind` looks a little different from the normal attribute. It looks like a Razor implicit expression. Yes, it is an implicit expression and `bind` is a reserved keyword. It is a directive attribute. The difference between a Razor directive and a Razor directive attribute is that the latter is used as the attribute of an HTML tag. The following are the directive attributes that we will use in this book:

- `@bind`: This is used in data binding
- `@on{EVENT}`: This is used in event handling
- `@on{EVENT} :preventDefault`: This is used to prevent the default action for the event
- `@on{EVENT} :stopPropagation`: This is used to stop event propagation
- `@ref`: This is used to provide a way to reference a component instance
- `@typeparam`: This is used to declare a generic type parameter

Now that we've learned about the basic syntax of the Razor markup language, let's move on to the real work.

Creating a Razor component

To develop a .NET MAUI Blazor app, we can choose to build all user interfaces using Blazor or a mix of Razor components with XAML components. We will start with the second option first since we already have a completed password manager app from *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*.

Redesigning the login page using a Razor component

The user interface that we want to replace first is the login page. We can use a Razor page to replace the XAML page to perform the same function.

In the Blazor Hybrid app, `BlazorWebView` is the control that hosts Razor components. We can change `LoginPage.xaml` 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:b="clr-namespace:Microsoft.AspNetCore.Components.
    WebView.Maui;
    assembly=Microsoft.AspNetCore.Components.WebView.Maui"
  xmlns:local="clr-namespace:PassXYZ.Vault.Pages"
  x:Class="PassXYZ.Vault.Views.LoginPage"
  Shell.NavBarIsVisible="False">
  <b:BlazorWebView HostPage="wwwroot/login.html"> ❶
    <b:BlazorWebView.RootComponents>
      <b:RootComponent Selector="#login-app" ❸
        ComponentType="{x:Type local:Login}" /> ❷
      </b:BlazorWebView.RootComponents>
    </b:BlazorWebView>
  </ContentPage>
```

The preceding page only contains a `BlazorWebView` control. Here, we can pay attention to the following items:

❶ The `HostPage` attribute is used to specify the HTML page to load in `BlazorWebView`. It is `login.html` (*Listing 7.2*) in this case.

The attributes of `RootComponent` define the Razor component and CSS selector to be used:

❷ The `ComponentType` attribute specifies the Razor Login component; we will work on it soon.

❸ The `Selector` attribute specifies the CSS selector on which our web UI will be loaded. We defined the CSS `#login-app` ID in `login.html`. The `login.html` HTML page is created and stored in the `wwwroot` folder. Let's review it in *Listing 7.2*:

Listing 7.2: `login.html` (<https://epa.ms/Login7-2>)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0, maximum-scale=1.0, user-scalable=no,
    viewport-fit=cover" />
  <title>PassXYZ.Vault Login</title>
  <base href="/" />
  <link rel="stylesheet" href="css/bootstrap/
    bootstrap.min.css" />
  <link href="css/app.css" rel="stylesheet" />
  <link href="PassXYZ.Vault.styles.css" rel="stylesheet" />
</head>

<body class="text-center">
  <div id="login-app">Loading...</div>
  <div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">X</a>
  </div>
```

❶

```
<script src="_framework/blazor.webview.js"
    autostart="false">
</script>
</body>
</html>
```

As we can see in *Listing 7.2*, it is very similar to `index.html`, which we discussed earlier. It defines the following CSS "login-app" ID ①, which is used to load our Razor component:

```
<div id="login-app">Loading...</div>
```

In the .NET MAUI Blazor template, the CSS framework, Bootstrap (`bootstrap.min.css`), is loaded by default. The embedded Bootstrap version was 5.1 at the time of writing. You may find a newer version in your project.

Bootstrap is a well-known framework for web development. It comes with many examples of how to use it. For the login page, there is a sign-in example on the Bootstrap website, as shown in *Figure 7.6*. We will use it to build our Login component:

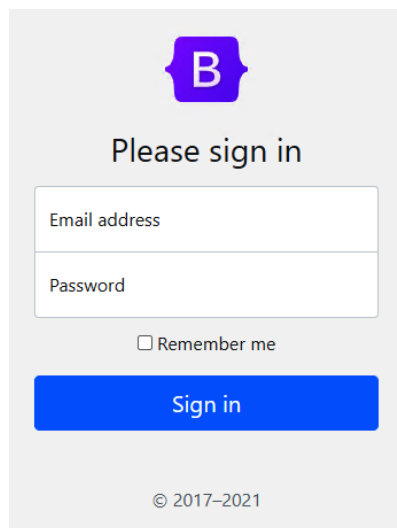


Figure 7.6: Bootstrap sign-in example

You can find this sign-in example at <https://getbootstrap.com/docs/5.1/examples/sign-in/>.

This sign-in example includes two files:

- `index.html` (Listing 7.3) is the user interface of the sign-in page. It defines the following:
 - Two `<input>` tags for the username ❶ and password ❷
 - An `<input>` tag ❸ for a checkbox to remember the username
 - A `<button>` tag ❹ to process the login activity
 - It uses Bootstrap CSS styles and its own styles defined in `signin.css`
- `signin.css` (Listing 7.4) defines the CSS styles specific to the sign-in page:

Listing 7.3: `index.html` (Bootstrap sign-in example)

```
<!doctype html>
<html lang="en">
  <head> ... </head>
  <body class="text-center">
    <main class="form-signin"> ❺
      <form>
        
        <h1 class="h3 mb-3 fw-normal">Please sign in</h1>
        <div class="form-floating">
          <input type="email" class="form-control"
            id="floatingInput" placeholder="name@example.com"> ❶
          <label for="floatingInput">Email address</label>
        </div>
        <div class="form-floating">
          <input type="password" class="form-control"
            id="floatingPassword" placeholder="Password"> ❷
          <label for="floatingPassword">Password</label>
        </div>
        <div class="checkbox mb-3">
          <label>
            <input type="checkbox" value="remember-me"> ❸
```

```

        Remember me
    </label>
</div>
<button class="w-100 btn btn-lg btn-primary"
    type="submit">Sign in</button>
    <p class="mt-5 mb-3 text-muted">&copy; 2017-2021</p>
</form>
</main>
</body>
</html>

```

In `signin.css` (Listing 7.4), we customize the `form-signin` CSS class ⑤, which is used in the sign-in page of `index.html`:

Listing 7.4: `signin.css` (Bootstrap sign-in example)

```

html,
body {
    height: 100%;
}
body {
    display: flex;
    align-items: center;
    padding-top: 40px;
    padding-bottom: 40px;
    background-color: #f5f5f5;
}
.form-signin {
    width: 100%;
    max-width: 330px;
    padding: 15px;
    margin: auto;
}
.form-signin .checkboxbox {
    font-weight: 400;
}

```

```

.form-signin .form-floating:focus-within {
    z-index: 2;
}

.form-signin input[type="email"] {
    margin-bottom: -1px;
    border-bottom-right-radius: 0;
    border-bottom-left-radius: 0;
}

.form-signin input[type="password"] {
    margin-bottom: 10px;
    border-top-left-radius: 0;
    border-top-right-radius: 0;
}

```

To create a new Razor component, we must create a folder called `Pages` in the project. After that, we can right-click on the `Pages` folder that we just created in Visual Studio and select **Add -> Razor Component...** We can name this component `Login.razor` and create it. After creating the file, we can copy the portion between the `<main>` tag from *Listing 7.3* into the Razor page inside a `<div>` tag, as shown in *Listing 7.5*:

Listing 7.5: Login.razor (<https://epa.ms/Login7-5>)

```

@using System.Diagnostics
@using PassXYZ.Vault.Services
@using PassXYZ.Vault.ViewModels
@Inject LoginViewModel viewModel ❶

<div>
    <main class="form-signin">
        <form>
            <img class="mb-4"...>
            <h1 class="h3 mb-3 fw-normal">Please sign in</h1>
            <div class="form-floating">
                <label for="floatingInput">Username</label>
                <input type="text" @bind="@currentUser.Username" ❷
                    class="form-control" id="floatingInput"
                    placeholder="Username">

```

```

    </div>
    <div class="form-floating">
        <label for="floatingPassword">Password</label>
        <input type="password" @bind="@currentUser
            .Password" class="form-control"
            id="floatingPassword" placeholder=
                "Password">
    </div>
    <div class="checkbox mb-3">
        <label>
            <input type="checkbox" value="remember-me">
                Remember me
        </label>
    </div>
    <button class="w-100 btn btn-lg btn-primary"
        type="submit" @onclick="OnLogin">Sign in</button>
    <p class="mt-5 mb-3 text-muted">&copy; 2017-2021</p>
</form>
</main>
</div>
@code {
    private LoginUser currentUser { get; set; } = default!; ❷
    protected override void OnInitialized() { ❸
        if (currentUser == null) {
            currentUser = viewModel.CurrentUser;
        }
    }
    private void OnLogin(MouseEventArgs e) {
        viewModel.OnLoginClicked(); ❺
    }
}

```

As we can see in `Login.razor`, Razor is a markup language that can mix HTML pages with C# code. We can embed C# code after the `@` symbol in HTML or put C# code inside the `@code` block.

In the `@code` block, we define a private variable called `currentUser` ❷ of the `LoginUser` type. We initialize `currentUser` in the override method, `OnInitialized()` ❸, to the property of the view model. We can refer to `currentUser` in HTML after the `@` symbol ❹. In the same way, we can define the `OnLogin()` event handler and refer to it in the `onclick` event.

Once the user enters their username and password, the properties of `currentUser` are filled in. When the user clicks the login button, `OnLogin()` will be invoked. The view model's `OnLoginClicked()` method ❺ is used to perform the login action.

The Model-View-ViewModel (MVVM) pattern in Blazor

The advantage of using Blazor for the user interface design is that we can do most of the UI design using HTML first. Once we are satisfied with the UI design, we can add our programming logic to the design. To separate the responsibilities in design, we can use the MVVM pattern, which we learned about in *Chapter 3, User Interface Design with XAML*, in Razor component development. In Blazor, we can treat the HTML markup as a view and the code block as a view model. If the logic in the code block is too complex, we can separate it into a C# code-behind file.

On the login page, we can still use `LoginViewModel` from the XAML world. This is because we will transit from Blazor back to the XAML UI inside `LoginViewModel`. This is done just to demonstrate how we can mix the Blazor and XAML UIs in one app. We will replace the XAML UI with the Blazor UI in the next chapter.

In a Razor component, we can either put both HTML and C# in one file, or we can split it into a Razor file and a C# code-behind file, just like XAML.

Let's do this for `Login.razor`. If we split it into two files, the component will be split into two partial classes residing in `Login.razor` and `Login.razor.cs`, as we can see in *Listing 7.6* and *Listing 7.7*:

Listing 7.6 Login.razor (<https://epa.ms/Login7-6>)

```
@namespace PassXYZ.Vault.Pages
<div>
    <main class="form-signin">
        <form>
            <img class="mb-4"...>
            <h1 class="h3 mb-3 fw-normal">Please sign in</h1>

            <div class="form-floating">
                <label for="floatingInput">Username</label>
```

```
        <input type="text" @bind="@currentUser.Username"
              class="form-control" id="floatingInput"
              placeholder="Username">
    </div>
    <div class="form-floating">
        <label for="floatingPassword">Password</label>
        <input type="password" @bind="@currentUser.
              Password" class="form-control"
              id="floatingPassword" placeholder="Password">
    </div>
    <div class="checkbox mb-3">
        <label>
            <input type="checkbox" value="remember-me"> Remember
me
        </label>
    </div>
    <button class="w-100 btn btn-lg btn-primary"
            type="submit" @onclick="OnLogin">Sign in</button>
    <p class="mt-5 mb-3 text-muted">&copy; 2021-2022</p>
</form>
</main>
</div>
```

In *Listing 7.6*, there is HTML markup only in `Login.razor`. This makes it look better in terms of separating the UI and logic. Let's review the corresponding C# code in *Listing 7.7*:

Listing 7.7: Login.razor.cs (<https://epa.ms/Login7-7>)

```
using Microsoft.AspNetCore.Components;
using System.Diagnostics;
using PassXYZ.Vault.Services;
using PassXYZ.Vault.ViewModels;
using Microsoft.AspNetCore.Components.Web;
namespace PassXYZ.Vault.Pages;

public partial class Login : ComponentBase {
    [Inject]
```

```
LoginViewModel viewModel { get; set; } = default!;  
private LoginUser currentUser { get; set; } = default!;  
  
protected override void OnInitialized() {  
    if (currentUser == null) {  
        currentUser = viewModel.CurrentUser;  
    }  
}  
private void OnLogin(MouseEventArgs e) {  
    viewModel.OnLoginClicked();  
}  
}
```

In *Listing 7.7*, we moved all the code from the `@code` block to the C# file inside the `Login` class, which is derived from the `ComponentBase` class. All Razor components are derived from `ComponentBase`.

You may have noticed that there is an attribute called `Inject` in the declaration of the `viewModel` property. This is used for dependency injection.

Dependency injection in Blazor

We introduced dependency injection in *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*. All the knowledge in that chapter applies equally here, but Blazor provides more. With Blazor, we can use dependency injection in both HTML and C#.

As shown in *Listing 7.5*, the following declaration is defined at the beginning of *Login.razor*:

```
@inject LoginViewModel viewModel
```

Here, we inject the `LoginViewModel` class into the `viewModel` variable. This is property injection, and we can use property injection much easier than before in Blazor.

To use dependency injection, we need to register `LoginViewModel` in `MauiProgram.cs`, as we did in *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*:

```
builder.Services.AddSingleton<LoginViewModel,  
    LoginViewModel>();
```

When we move it to the C# code-behind file, we can use the `Inject` attribute to do the same:

```
[Inject]  
LoginViewModel viewModel { get; set; } = default!;
```

In web development, we usually use both HTML and CSS styles together to design web user interfaces. In the Bootstrap example, there is a `signin.css` file. Where do we keep our CSS styles? Let's look at this topic in the next section.

CSS isolation

When we introduced the Bootstrap sign-in example, we mentioned that there's an HTML file and a CSS file. Where do we put that CSS file so that we can reuse the sign-in CSS styles on our login page?

In HTML design, when we use a CSS framework such as Bootstrap, sometimes, we also need to customize styles at the page level. To support this in Blazor, there is a technique called CSS isolation for Razor components. For component- or page-specific CSS styles, we can keep it in a file with the `.razor.css` extension. The filename should match the name of the `.razor` file in the same folder. For our login page, we can copy `sign-in.css` from the Bootstrap example to `Login.razor.css` and make minor modifications, as shown in *Listing 7.8*:

Listing 7.8: Login.razor.css (<https://epa.ms/Login7-8>)

```
div {
    display: flex;
    align-items: center;
    background-color: #f5f5f5;
}
.form-signin {
    width: 100%;
    max-width: 330px;
    padding: 15px;
    margin: auto;
}
.form-signin .checkbox {
    font-weight: 400;
}
.form-signin .form-floating:focus-within {
    z-index: 2;
}
.form-signin input[type="email"] {
    margin-bottom: -1px;
    border-bottom-right-radius: 0;
    border-bottom-left-radius: 0;
```

```
}  
.form-signin input[type="password"] {  
    margin-bottom: 10px;  
    border-top-left-radius: 0;  
    border-top-right-radius: 0;  
}
```

The styles defined in `Login.razor.css` are only applied to the rendered output of the `Login` component. Finally, let's look at this new login UI in Blazor:

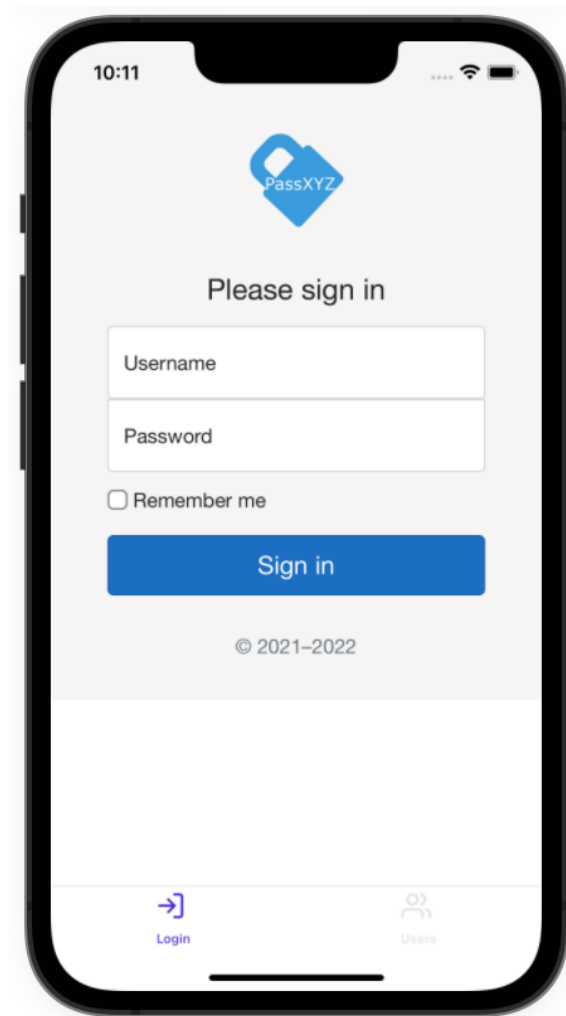


Figure 7.7: Login page

We can see that the look and feel of this new UI in *Figure 7.7* (shown in iOS) are the same as those from the Bootstrap sign-in example, except we changed the icon. The functionality of logging in hasn't changed, except we used Blazor to create a new UI. After we log in using this Razor page, the remaining programming logic is still the same as what was shown in *Chapter 6, Introducing Dependency Injection and Platform-Specific Services*. The UI framework is switched back to XAML after login since we haven't changed anything after that yet.

Summary

In this chapter, we learned about Blazor and how to develop a Blazor Hybrid app. Blazor is an alternative solution for UI design for .NET MAUI. The difference between Blazor and XAML is that the look and feel of the XAML user interface are the same as the native user interface, but the Blazor UI has the look and feel of a web app. In terms of the functionalities that they can provide, there is not much difference. We can mix Blazor and XAML in one app. We can use the MVVM pattern in both XAML and Blazor. With Blazor, the UI code can be shared between the Blazor Hybrid app and the web app. If you are looking for a solution that supports both native and web apps, .NET MAUI Blazor could be a good choice.

In the next chapter, we will replace all the user interfaces in our app using Blazor. We will also introduce how to do the initial UI design using layout and routing.

8

Understanding the Blazor Layout and Routing

In the last chapter, we learned how to design a login page using Blazor. The app layout and navigation hierarchy are still XAML-based. Our app is using a mixed UI implementation of Blazor and XAML. Blazor is a different choice of UI design for a .NET MAUI app. In this (second) part of the book, we will rebuild the entire UI using Blazor. The first step of UI design usually starts from the implementation of the layout and navigation, so in this chapter, we will introduce the layout and routing of Blazor.

We will cover the following topics in this chapter:

- Blazor routing
- Using Blazor layout components

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to *Development environment setup* in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code for this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter08>

The source code can be downloaded using the following Git command:

```
git clone -b chapter08 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

Understanding client-side routing

The routing and layout of Blazor are similar to the concept of Shell and navigation in the XAML world. In *Chapter 5, Navigation using .NET MAUI Shell and NavigationPage*, when we introduced navigation and Shell, we discussed the routing strategy of Shell. Shell provides a URI-based navigation experience that uses routes to navigate to the selected pages. The routing of Blazor is quite similar to that.

The routing of Blazor provides a way to switch from one Razor page to another. The rendering of Razor pages in BlazorWebView is similar to web apps running in a browser.

In a classic web application, when we load an HTML page in a browser, the HTML page is retrieved from the web server. When we choose a different route, we load a new page from the server. For **Single-Page Applications (SPAs)**, things work a little differently.

Blazor WebAssembly apps are SPAs. When the app is started, the app is loaded in a browser. After that, the navigation of pages only happens on the client side. This is so-called client-side routing. Blazor Hybrid apps use client-side routing as well.

Setup of BlazorWebView

To perform client-side routing, the router has to be installed at the application startup. In .NET MAUI apps, the entry point of both XAML and Blazor is set up in `App.xaml.cs`. We can refer to changing `App.xaml.cs` here (in the code) to switch the UI implementation from XAML to Blazor.

The Blazor Hybrid app runs inside BlazorWebView. To start the Blazor Hybrid app, we need to set up an instance of BlazorWebView first. In the last chapter, we set it up in `LoginPage` and we navigated back to Shell after logging in successfully.

To set up an instance of BlazorWebView for the entire application, we need to replace the instance assigned to the `MainPage` property of the `App` class. To do this, we changed the constructor of the `App` class (in `App.xaml.cs`) as follows:

```
public App()
{
    InitializeComponent();
    #if MAUI_BLAZOR
        MainPage = new MainPage();
    #else
        Routing.RegisterRoute(nameof(ItemsPage),
            typeof(ItemsPage));
        Routing.RegisterRoute(nameof(ItemDetailPage),
            typeof(ItemDetailPage));
        Routing.RegisterRoute(nameof(NewItemPage),
```

1

```

        typeof(NewItemPage));

    MainPage = new AppShell();
#endif
}

```

❶ We can define a symbol, `MAUI_BLAZOR`, to set up conditional compilation so that we can switch between XAML and Blazor UI in the build. To use Blazor UI, we set the `MainPage` property to a `MainPage` instance. In the `MainPage` class, we define the `BlazorWebView` control as follows:

```

<BlazorWebView HostPage="wwwroot/index.html">
  <BlazorWebView.RootComponents>
    <RootComponent Selector="#app" ComponentType="{x:Type
      local:Main}" />
  </BlazorWebView.RootComponents>
</BlazorWebView>

```

In `BlazorWebView`, it loads an HTML page (`index.html`) to start the Blazor UI setup. Let's see how the Router setup works.

Setup of Router

Blazor UI is an HTML page-based UI design. It is similar to an SPA and starts from a static HTML page. In `BlazorWebView`, the HTML page to be loaded is `index.html`, which is very similar to the `login.html` page that we introduced in the last chapter. The top-level Razor component that is loaded in `RootComponent` is the `Main` component that we can see here:

Listing 8.1: Main.razor (<https://epa.ms/Main8-1>)

```

<Router AppAssembly="@typeof(Main).Assembly">                                ❶
  <Found Context="routeData">                                              ❷
    <RouteView RouteData="@routeData"
      DefaultLayout="@typeof(MainLayout)" />
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
  </Found>
  <NotFound>                                                                ❸
    <LayoutView Layout="@typeof(MainLayout)">
      <p role="alert">Sorry, there's nothing at this
        address.</p>

```

```
    </LayoutView>  
  </NotFound>  
</Router>
```

As we can see in *Listing 8.1*, we set up the Router component, ①, in `Main.razor`.

In the Router component, it uses reflection to scan all the page components to build a routing table. The `AppAssembly` parameter specifies the assemblies to be scanned.

If there is a navigation event, the router checks the routing table for a matching route. The Router component is a templated component. We will discuss what a templated component is in a later chapter. When a route is found, the `Found` template is used. Otherwise, the `NotFound` template is used when there are no matching routes.

The `Found` template, ②, uses a `RouteView` component to render the selected component with its layout. The layout is specified in the `DefaultLayout` attribute. We will discuss the layout in the next section. The new page to be loaded, along with any route parameters, is passed using an instance of the `RouteData` class.

If a match could not be found, the `NotFound` template, ③, is rendered. The `NotFound` template uses a `LayoutView` component to display error messages. The layout used by `LayoutView` is specified using a `Layout` attribute.

Defining routes

Once we have set up the router, we can create pages and define route templates in pages. The router will scan the route templates defined in pages to build a routing table.

At a high level, we can create the navigation hierarchy and route templates of our app referring to *Figure 8.1*.

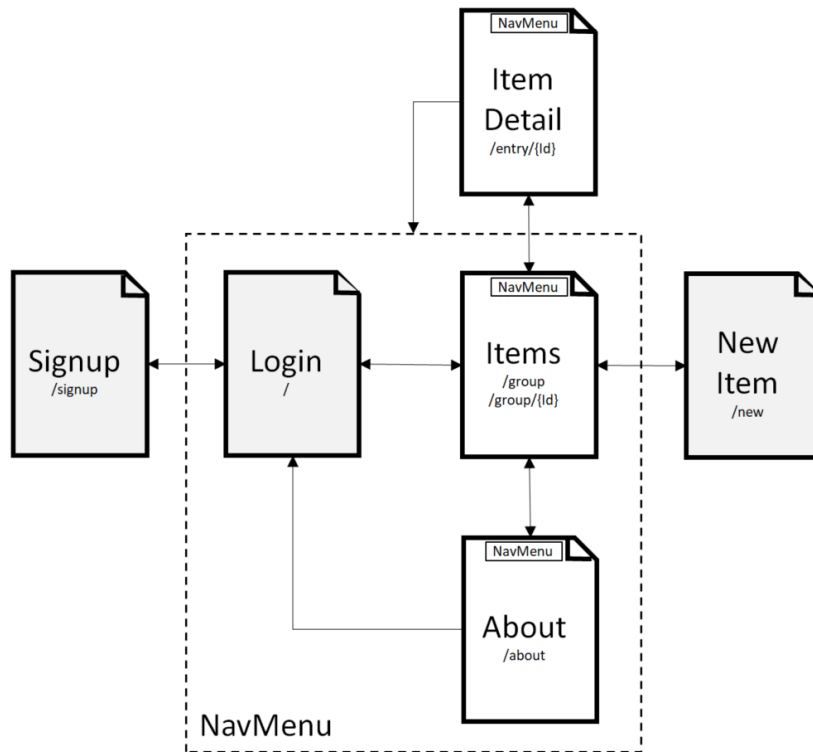


Figure 8.1: Navigation hierarchy of Razor pages

We listed the major pages in our app in *Figure 8.1*. Each page has a name that is also the class name of a Razor page. The path under the name is the route template. For example, for the *About* page, we can declare the route template as follows:

```
@page "/about"
```

The `@page` directive includes two parts – the directive name, and the route template. In this example, the route template is `"/about"`, which must be in quotes and always starts with a forward slash (`/`). Since the final output of a Razor page is an HTML page, we can navigate to a Razor page just like a web page using an anchor tag, `<a>`, as shown here:

```
<a href="/about">About</a>
```

Passing data using route parameters

When we navigate to a page using a route template, we can pass data to the page using route parameters. If we recall how to pass data with the query parameter in *Shell*, the usage of the route parameter is similar to the query parameter.

As we can see in *Figure 8.1*, after we log in successfully, the `Items` page is shown and a list of items in the root group is displayed as in *Figure 8.2*. On this page, if we click on an item, we can navigate to the selected item based on the item type. To find the selected item, an `itemId` value is passed to the new page as a parameter.

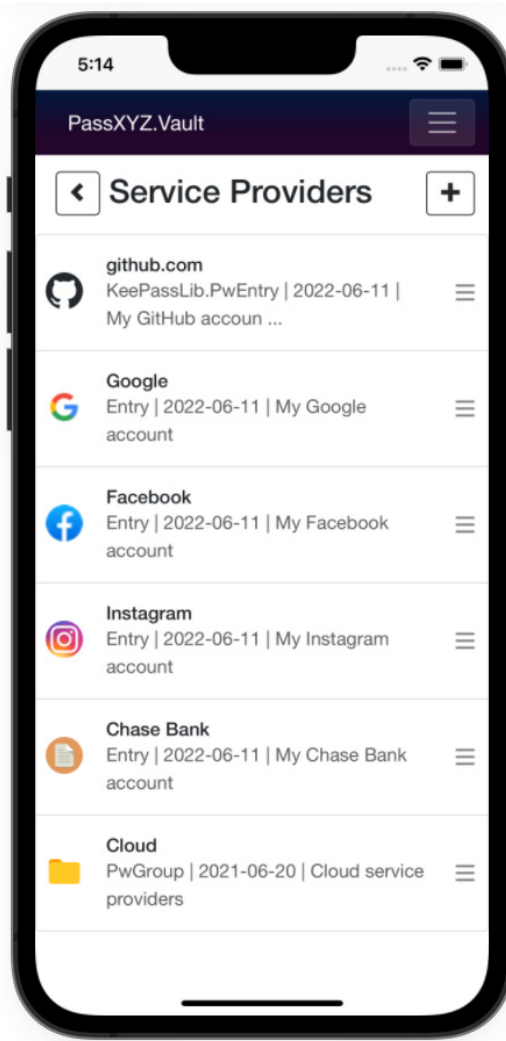


Figure 8.2: Items page in a Blazor Hybrid app

In the `Items` page, we have the following route templates defined:

```
@page "/group"  
@page "/group/{SelectedItemId}"
```

The first route template is used when we display the root page. The second one is used when we select a group. The group `Id` value is passed to the `Items` page using the `SelectedItemId` route parameter.

To specify the type of route parameter, we can add constraints to it with the data type as shown here:

```
@page "/user/{Id:int}"
```

In the preceding page directive, we specify the data type of `Id` as an integer. Please refer to the corresponding Microsoft documents to find more details about route constraints. You can find the relevant document here:

<https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing?view=aspnetcore-6.0#route-constraints>

Navigating with NavigationManager

In a Razor page, although we can generally navigate to another page using an `<a>` anchor tag, sometimes, we might need to do it using code. For example, when we handle an event, we may redirect to a page in the event handler. This is exactly the case on our `Login` page. Let's see how to navigate to the `Items` page using `NavigationManager` after logging in successfully.

In our app, we need to redirect to the `Items` page to display the root group after login. The UI of the `Login` page is the same as the one in the last chapter, but we changed the event handle in `Login.razor.cs` to the following one:

```
namespace PassXYZ.Vault.Pages;

public partial class Login : ComponentBase {
    [Inject]
    private IUserService<User> userService { get; set; } =
        default!;
    [Inject]
    private IDataStore<Item> dataStore { get; set; } =
        default!;
    [Inject]
    private NavigationManager navigationManager {get; set;} ❶
    private LoginUser currentUser => LoginUser.Instance;

    private async void OnLogin(MouseEventArgs e) {
        bool status = await userService.LoginAsync
            (currentUser);
        if (status) {
```

```

        string path = Path.Combine(PxDataFile.TmpFilePath,
            currentUser.FileName);
        if (File.Exists(path)) {
            bool result = await dataStore.MergeAsync(path);
        }
        navigationManager.NavigateTo("/group");
    }
}

```

❶ We get an instance of `NavigationManager` using dependency injection.

❷ We call the `NavigateTo("/group")` method of `NavigationManager` to navigate to the `Items` page.

We learned how to use routing and navigation in this section. In the next step, we can implement a navigation hierarchy similar to what we did with Shell navigation with Blazor UI.

The top level of the HTML page navigation hierarchy includes a header, toolbar, menu, and footer. We can implement the layout using a Blazor layout component. It is something similar to the flyout and menu items in Shell, which we introduced in *Chapter 5, Navigation using .NET MAUI Shell and NavigationPage*.

Using Blazor layout components

Most web pages usually contain fixed parts, such as a header, footer, or menu. We can design a page using a layout together with the page content to reduce the redundant code. The page itself contains the content that we want to show the users and the layout helps to build the styles and provide navigation methods.

Blazor layout components are a class derived from `LayoutComponentBase`. Anything we can do with a regular Razor component we can do to the layout components as well.

In *Listing 8.1*, we can see that `MainLayout` is used as the default layout of the pages. It is defined in *Listing 8.2* here:

Listing 8.2: `MainLayout.razor` (<https://epa.ms/MainLayout8-2>)

```

@inherits LayoutComponentBase
<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>
    <div class="main">
        @Body
    </div>
</div>

```

```
</div>

<main>
  @Body
</main>
</div>
```

3

In the `MainLayout` component, **1**, it inherits the `LayoutComponentBase` class. **2** It includes a `NavMenu` component to define the menu for navigation. **3** Inside the `<main>` tag, it uses the `@Body` Razor syntax to specify the location in the layout markup where the content is rendered.

Let's review the `NavMenu` component in detail since it is the top-level navigation method in our app. Please refer to *Figure 8.3* to see the UI of `NavMenu` before we review the code. `NavMenu` includes three menu items **Home**, **About**, and **Logout**.

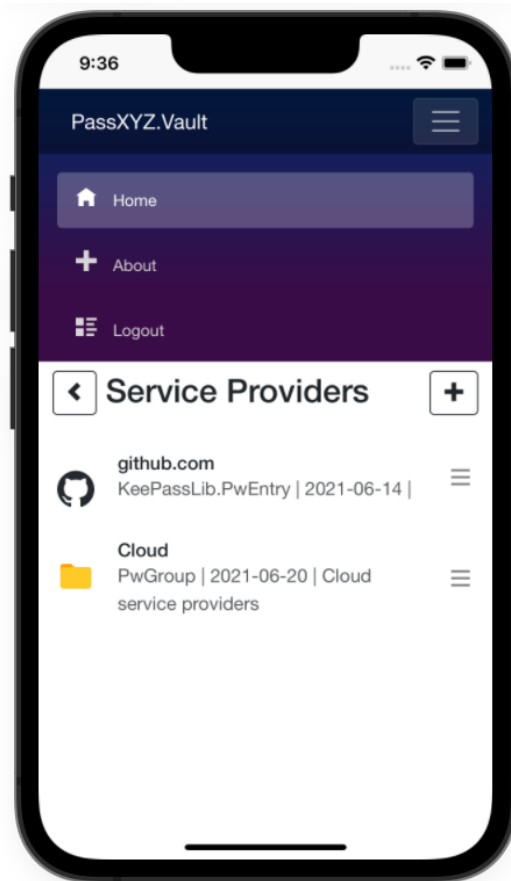


Figure 8.3: `NavMenu`

NavMenu is a Razor component that defines the links for navigation. We can review the source code of NavMenu in *Listing 8.3* here:

Listing 8.3: NavMenu.razor (<https://epa.ms/NavMenu8-3>)

```

<div class="top-row ps-3 navbar navbar-dark"> ①
  <div class="container-fluid">
    <a class="navbar-brand" href="">PassXYZ.Vault</a>
    <button title="Navigation menu" class="navbar-toggler"
      @onclick="ToggleNavMenu"> ②
      <span class="navbar-toggler-icon"></span>
    </button>
  </div>
</div>

<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
  <nav class="flex-column">
    <div class="nav-item px-3"> ③
      <NavLink class="nav-link" href="/group"> ④
        <span class="oi oi-home" aria-hidden="true"></span>
        Home
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="/about">
        <span class="oi oi-plus" aria-hidden="true"></span>
        About
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="" Match=
        "NavLinkMatch.All">
        <span class="oi oi-list-rich" aria-hidden="true">
          </span>
          Logout
        </NavLink>

```

```

    </div>
  </nav>
</div>

@code {
    private bool collapseNavMenu = true;

    private string NavMenuCssClass => collapseNavMenu ?
        "collapse" : null;

    private void ToggleNavMenu() {
        collapseNavMenu = !collapseNavMenu;
    }
}

```

In the source code of the `NavMenu` component, we can see that it is a Bootstrap `navBar` component with some C# logic in the code block. ① `NavBar` is defined using a `navbar` Bootstrap class in the `<div>` tag as follows:

```
<div class="top-row ps-3 navbar navbar-dark">
```

As we can see in *Figure 8.3*, there is a hamburger icon, ②, in the top left of the screen using a `<button>` tag to toggle `NavMenu`. The hamburger button UI is implemented using the `navbar-toggler` Bootstrap class as follows:

```

<button title="Navigation menu" class="navbar-toggler"
    @onclick="ToggleNavMenu">
    <span class="navbar-toggler-icon"></span>
</button>

```

There are three links defined as the menu items using the `nav-item` Bootstrap class, ③. The link is defined using `NavLink` ④ instead of the anchor tag, `<a>`. The `NavLink` component behaves like `<a>` except it toggles an active CSS class based on whether its `href` matches the current URL as we can see here:

```

<div class="nav-item px-3">
    <NavLink class="nav-link" href="/group">
        <span class="oi oi-home" aria-hidden="true"></span>
        Home
    </NavLink>
</div>

```

```
    </NavLink>
  </div>
```

We explained `MainLayout`, which is the default layout in our app. Let us look at how to apply the layout to a component.

Applying a layout to a component

`MainLayout` is used as the default layout component, so it will apply to all pages if we don't specify a layout. Sometimes, we need to use a specific layout instead of the default layout. For example, in our app, we use a different layout component on the `Login` page instead of the default layout (see *Listing 8.4*). `MainLayout` includes a `NavMenu` component. We do not want to show it on the `Login` page since we don't allow the user to see any other content before logging in. Let us look at the changes to the `Login` page after we apply a specific layout in *Listing 8.4*:

Listing 8.4: `Login.razor` (<https://epa.ms/Login8-4>)

```
@page "/"
@layout LogoutLayout
@namespace PassXYZ.Vault.Pages

<div class="text-center">
  <main class="form-signin">
    <form>
      <img id="first" class="mb-4" src=
        "passxyz-blue.svg"...>
      <h1 class="h3 mb-3 fw-normal">Please sign in</h1>
      <div class="form-floating">
        <label for="floatingInput">Username</label>
        <input type="text"
          @bind="@currentUser.Username"...>
      </div>
      <div class="form-floating">
        <label for="floatingPassword">Password</label>
        <input type="password" @bind=
          "@currentUser.Password"...>
      </div>
      <div class="checkbox mb-3">
```

1

```
        <label>
            <input type="checkbox" value="remember-me">
                Remember me
        </label>
    </div>
    <button...>Sign in</button>
    <p class="mt-5 mb-3 text-muted">&copy; 2021-2022</p>
</form>
</main>
</div>
```

To use a specific layout, we can use the `@layout` Razor directive, ❶. In the Login page, we use the `LogoutLayout` layout. The code of `LogoutLayout` is shown in *Listing 8.5* here:

Listing 8.5: LogoutLayout.razor (<https://epa.ms/LogoutLayout8-5>)

```
@inherits LayoutComponentBase
<div class="page">
    <main>
        <div class="top-row px-4">
            <a href="#" target="_blank">Sign-in</a>
        </div>
        <article class="content px-4">
            @Body
        </article>
    </main>
</div>
```

In `LogoutLayout`, we removed the `NavMenu` element and added a sign-in link to allow a new user to sign up.

Nesting layouts

Layout components can also be nested. In `MainLayout`, we did not specify any margin for the content. `MainLayout` is suitable for a content list view on an items page or item details page. However, it does not look good for content pages, such as the About page. We can use a different layout for an About page and this layout is nested in `MainLayout`. We can call it `PageLayout` and we can see an implementation in *Listing 8.6*:

Listing 8.6: `PageLayout.razor` (<https://epa.ms/PageLayout8-6>)

```
@inherits LayoutComponentBase
@layout MainLayout

<article class="content px-4">
    @Body
</article>
```

`PageLayout` is a layout component that uses `MainLayout`. It puts `@Body` in an `<article>` tag with the `"content px-4"` style applied so that the content can apply the style that is suitable for a paragraph of text.

In the About page, we can set the layout to `PageLayout` like so:

```
@page "/about"
@layout PageLayout
```

We have now introduced the routing and layout of Blazor. With this knowledge, it's time to implement the navigation elements of our app.

Implementing navigation elements

In *Chapter 5, Navigation using .NET MAUI Shell and NavigationPage*, when we introduced Shell, we mentioned the absolute route and relative route in Shell. We can define absolute routes in a visual navigation hierarchy and navigate to the relative route through query parameters.

This navigation strategy is similar in the Blazor version of our app. As we can see in *Figure 8.4*, we implement Blazor UI elements in the same way as our XAML version.

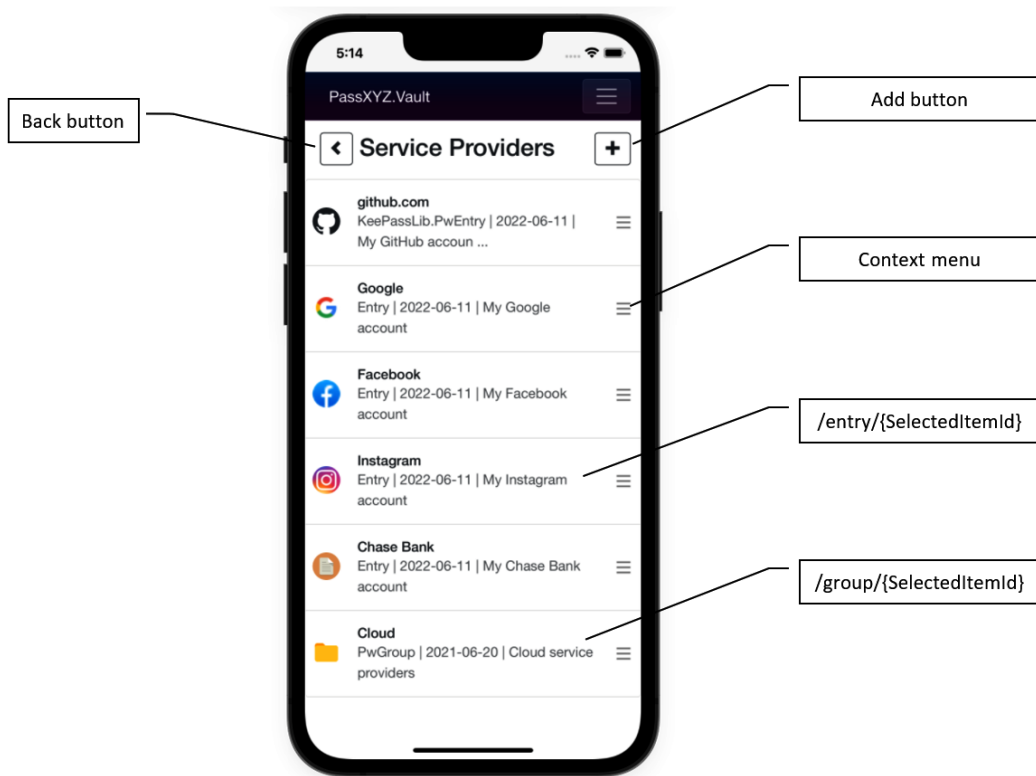


Figure 8.4: Navigation elements

The `Items` page is the main page of our app after login. In the `Items` page, in which the list of items is displayed, the following UI elements are related to the navigation:

- A list view – The user can navigate through the list and select an item.
- Context menu – It is associated with each item in the list view. The user can edit or delete an item using the context menu.
- Back button – The user can use it to navigate back.
- Add button – The user can use it to add new items.

In this section, we will implement the preceding navigation elements using the knowledge we have gained.

Implementing a list view

In the XAML version, our navigation starts from a list of items after the user logs in to the app. The list view is implemented using a .NET MAUI `Listview` control, which uses the underlying platform-specific UI component, so the look and feel are the same as the platform-specific one. In the Blazor version, we use a web UI, so the look and feel are the same on different platforms.

To implement a list view using a web UI, we have many choices. In this book, we stick with the Bootstrap framework. The way that we are going to do so is the same as in the last chapter. We can reuse the UI design from the Bootstrap examples. We are using Bootstrap 5.1 in this book, so we can refer to the following list group example shown in *Figure 8.5*.

Buttons

- Button group
- Card
- Carousel
- Close button
- Collapse
- Dropdowns
- List group**
- Modal
- Navs & tabs
- Navbar
- Offcanvas
- Pagination
- Placeholders
- Popovers
- Progress
- Scrollspy
- Spinners
- Toasts
- Tooltips

Custom content

Add nearly any HTML within, even for linked list groups like the one below, with the help of [flexbox utilities](#).

List group item heading 3 days ago
 Some placeholder content in a paragraph.
 And some small print.

List group item heading 3 days ago
 Some placeholder content in a paragraph.
 And some muted small print.

List group item heading 3 days ago
 Some placeholder content in a paragraph.
 And some muted small print.

```

<div class="list-group">
  <a href="#" class="list-group-item list-group-item-action active" aria-current="true">
    <div class="d-flex w-100 justify-content-between">
      <h5 class="mb-1">List group item heading</h5>
      <small>3 days ago</small>
    </div>
    <p class="mb-1">Some placeholder content in a paragraph.</p>
    <small>And some small print.</small>
  </a>
  <a href="#" class="list-group-item list-group-item-action">

```

Copy

Figure 8.5: Bootstrap list group

The preceding example can be found at the following URL:

<https://getbootstrap.com/docs/5.1/components/list-group/>

A Bootstrap list group can be used to build a UI-like `ListView` component in XAML. To do this, we can apply a CSS class, `list-group`, to HTML tags, such as `<ul>` or `<div>`, to create a list group. Inside the list group, the `list-group-item` CSS class is applied to the list item in the group.

In the XAML version, we support CRUD operations using the context menu. However, there is no context menu available in the Bootstrap list group, so we need to implement a context menu by ourselves. To implement a context menu in the list group, we can use the dropdown component of Bootstrap.

To use the dropdown component, we need to include the JavaScript dependency in `index.html` as follows:

```
<script src="_framework/blazor.webview.js"
    autostart="false"></script>
<script src="css/bootstrap/bootstrap.bundle.min.js">
    </script>
```

We added a JavaScript file, `bootstrap.bundle.min.js`, after `blazor.webview.js`. The `bootstrap.bundle.min.js` JavaScript file is part of the Bootstrap release package.

To create a new Razor component, `Items`, we can right-click on the **Pages** folder in Visual Studio and select **Add -> Razor Component...** to create it. We add the following code in *Listing 8.7* and name the Razor file `Items.razor`:

Listing 8.7: `Items.razor` (<https://epa.ms/Items8-7>)

```
@page "/"group"
@page "/"group/{SelectedItemId}"

<!-- Back button and title -->
<div class="container">...
<!-- List view with context menu -->
<div class="list-group">
    @foreach (var item in items) {
        <div class="dropdown list-group-item
            list-group-item-action...>
            
            <a href="@item.GetActionLink()"
                class="list-group-item...>
                <div class="d-flex">
                    <div>
```

1**2**

```

        <h6 class="mb-0">@item.Name</h6>
        <p class="mb-0 opacity-75">@item.Description
        </p>
    </div>
</div>
</a>
<button class="opacity-50 btn btn-light
    dropdown-toggle"
    type="button" id="itemsContextMenu"
    data-bs-toggle="dropdown" aria-expanded="false">
    <span class="oi oi-menu" aria-hidden="true"></span>
</button> ❸
<ul class="dropdown-menu"
    aria-labelledby="itemsContextMenu"> ❹
    <li>
        <button class="dropdown-item"
            data-bs-toggle="modal"
            data-bs-target="#editModel"> Edit </button>
    </li>
    <li>
        <button class="dropdown-item"
            data-bs-toggle="modal"
            data-bs-target="#deleteModel"> Delete </button>
    </li>
</ul>
</div>
}
</div>
<!-- Editing Modal -->
<div class="modal fade" id="editModel" tabindex="-1"
    aria-labelledby="editModelLabel" aria-hidden="true">...
<!-- Deleting Modal -->
<div class="modal fade" id="deleteModel" tabindex="-1"
    aria-labelledby="deleteModelLabel" aria-hidden="true">...
<!-- New Modal -->

```

```
<div class="modal fade" id="newItemModel" tabindex="-1" aria-
labelledby="newItemModelLabel" aria-hidden="true">...
```

In `Items.razor`, ❶, we can copy Bootstrap list group sample code that uses the `<div>` tag by applying the `list-group` CSS class to it.

❷ We customize the **list group item** to meet our requirements as you can see in *Figure 8.6*. The list group item is created inside a `foreach` loop using the `<div>` tag, which includes an icon, a name, a description, and a context menu:

```
<div class="dropdown list-group-item list-group-item-action...">
```

We apply `dropdown`, `list-group-item`, and `list-group-item-action`, CSS classes, to the `<div>` tag, so it is a list group item including a context menu using a dropdown.

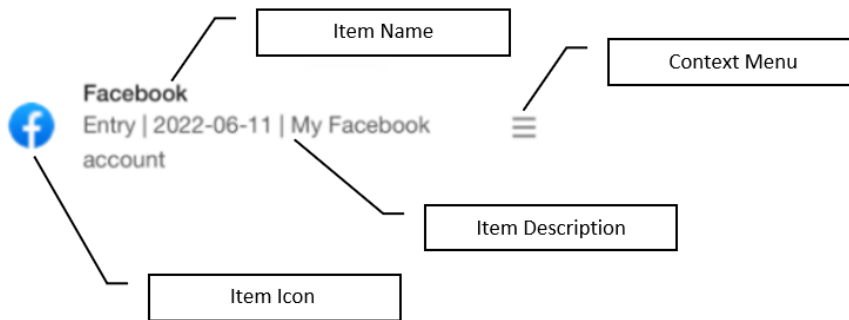


Figure 8.6: List group item

Inside the list group item, we use an `<img>` tag to display the item icon:

```

```

We can get the icon source using an extension method, `GetIcon()`, from the `Item` class. To create the extension method, we add a new class file under the `Shared` folder and name it `ItemEx.cs` as shown in *Listing 8.8*.

An `<a>` anchor tag is used to display the name and description of the item. Inside `<a>`, the name and description are defined like so:

```
<a href="@item.GetActionLink()" class=
    "list-group-item...">
    <div class="d-flex">
        <div>
            <h6 class="mb-0">@item.Name</h6>
```

```

        <p class="mb-0 opacity-75">@item.Description
    </p>
</div>
</div>
</a>

```

We can get the link of the item using the `GetActionLink()` extension method, which is also defined in *Listing 8.8*.

The context menu is a Bootstrap dropdown component including a `<button>` tag, ❸, and an unordered list using the `<ul>` tag, ❹. This button is displayed as a hamburger icon using Open Iconic font.

Open Iconic icons

We use Open Iconic icons in Blazor UI design. Open Iconic is an open source icon set with 223 icons in SVG, web font, and raster formats. In XAML design, we use FontAwesome, which can also be used in Blazor with Bootstrap. However, we need extra configuration before we can use it. Open Iconic is included in the Blazor template together with Bootstrap. We can use it directly without any extra configuration. For example, to display a hamburger icon in the context menu, we can use the following HTML tag:

```
<span class="oi oi-menu" aria-hidden="true"></span>
```

In the drop-down menu, there are two context action buttons, Edit and Delete. We apply the `dropdown-item` CSS class to the buttons. The context action button triggers a dialog box to perform the CRUD operations so that there are two Bootstrap modal CSS attributes, `data-bs-toggle` and `data-bs-target`, applied to it. We will discuss how to handle CRUD operations in the next chapter.

Let us review the extension methods of `Item` that we will use to support a list view UI in *Listing 8.8*:

Listing 8.8: ItemEx.cs (<https://epa.ms/ItemEx8-8>)

```

using KeePassLib;
using KPCLib;
using PassXYZLib;

namespace PassXYZ.Vault.Shared {
    public static class ItemEx {
        public static string GetIcon(this Item item) {           ❶
            if (item.IsGroup) {
                // Group
            }
        }
    }
}

```

```

        if(item is PwGroup group) {
            if(group.CustomData.Exists(PxDefs.PxCustomDataIconName)) {
                return $"/images/{group.CustomData.Get
                    (PxDefs.PxCustomDataIconName)}";
            }
        }
    }
else {
    // Entry
    if(item is PwEntry entry) {
        if(entry.CustomData.Exists
            (PxDefs.PxCustomDataIconName)) {
            Return $"/images/{entry.CustomData.Get
                (PxDefs.PxCustomDataIconName)}";
        }
    }
}
// 2. Get custom icon
return item.GetCustomIcon();
}

/// <summary>
/// Get the action link of an item.
/// </summary>
public static string GetActionLink(this Item
    item, string? action = default) {
    string itemType = (item.IsGroup) ?
        PxConstants.Group : PxConstants.Entry;

    return (action == null) ? $"/{itemType}/{item.Id}" :
        $"/{itemType}/{action}/{item.Id}";
}
/// <summary>
/// Get the parent link of an item.
/// </summary>
public static string? GetParentLink(this Item item) {

```

```

        Item? parent = default;
        if (item == null) return null;
        if (item.IsGroup) {
            PwGroup group = (PwGroup) item;
            if (group.ParentGroup == null) return null;
            parent = group.ParentGroup;
        }
        else {
            PwEntry entry = (PwEntry) item;
            if (entry.ParentGroup == null) return null;
            parent = entry.ParentGroup;
        }
        return $"{PxConstants.Group}/{parent.Id}";
    }
}

```

In *Listing 8.8*, we define a static class, `ItemEx`, to implement an extension method of the `Item` class. In this class, we defined three extension methods to get the URL needed for navigation:

- `GetIcon()`, ① – returns the URL of the icon image
- `GetActionLink()`, ② – return the URL of a selected item based on the item type
- `GetParentLink()`, ③ – returns the URL of the parent item

With the preceding implementation of a list view UI, we have a list, including password entries and groups. When an item is selected, we actually click an anchor tag, `<a>`. The `href` property of `<a>` is set to the return value of the `GetActionLink()` method. The return value of this method is in the format of a `"/{itemType}/{item.Id}"` route template, which can be used to navigate to the selected item. At the right-hand side of each item, there is a context menu button. If we click on it, a list of context actions is shown, and we can select an action to edit or delete the current item.

We can handle most of the navigation actions now, but there are still two actions missing. When we enter a child group, we cannot navigate back, and we also cannot add a new item. We will add these two functions in the next section.

Adding a new item and navigating back

To support navigating back and adding a new item, we can add a **Back** button and an **Add** button in the title bar to simulate the navigation page of the XAML version as shown in *Figure 8.7*:

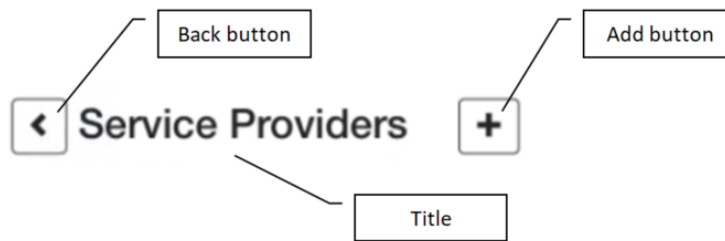


Figure 8.7: The Title bar of the Items page

As we can see in the title bar here, three UI elements are included:

- **Title** – the title of the current item group
- The **Back** button – the button can be used to navigate back, but it won't be shown when there is no parent group
- The **Add** button – the button can be used to add a new item

To review the implementation, we can expand the code of the **Back** button and the **Title** part in *Listing 8.7* as shown here:

```
<!-- Back button and title -->
<div class="container">
  <div class="row">
    <div class="col-12">
      <h1>
        @if (selectedItem!.GetParentLink() != null) {
          <a class="btn btn-outline-dark" href=
            "@selectedItem!.GetParentLink()"><span
              class="oi oi-chevron-left"
              aria-hidden="true"></span></a> ❶
        }
        @(" " + Title)
        <button type="button" class="btn btn-outline-dark
          float-end" data-bs-toggle="modal" data-bs-
            target="#newItemModel"><span class="oi
              oi-plus" aria-hidden="true"></span></button> ❷
      </h1>
    </div>
```

```

    </div>
</div>

```

The **Back** button, ❶, is implemented as an anchor tag, `<a>`. The `href` attribute of the anchor tag is set to the return value of the `Item` extension method, `GetParentLink()`. This function returns the link of the parent item in the route template format, so we can navigate back using this link. If there is no parent group, such as a root group, the **Back** button is not shown.

The **Add** button, ❷, is implemented using a `<button>` tag. The **Add** button is displayed on the right-hand side of the title bar. To push this button to the right of the screen, we can use a Bootstrap class, `float-end`. When the user clicks on this button, a new item dialog box is shown. The dialog box is set using the following attributes:

```
data-bs-toggle="modal" data-bs-target="#NewItemModel"
```

There are three Bootstrap modal dialogs used in `Items.razor` as shown in *Listing 8.7*:

```

<!-- Editing Modal -->
<div class="modal fade" id="editModel" tabindex="-1"
    aria-labelledby="editModelLabel" aria-hidden="true">...
<!-- Deleting Modal -->
<div class="modal fade" id="deleteModel" tabindex="-1"
    aria-labelledby="deleteModelLabel" aria-hidden="true">...
<!-- New Modal -->
<div class="modal fade" id="NewItemModel" tabindex="-1" aria-
    labelledby="NewItemModelLabel" aria-hidden="true">...

```

We use these dialogs to perform CRUD operations. To implement these dialogs, we also reuse code from Bootstrap. It is quite straightforward to do it this way, but there is a lot of duplicated code involved. To save space, they are collapsed in *Listing 8.7*. In the next chapter, we will explain the implementation of model dialogs and convert the code into reusable Razor components.

Summary

In this chapter, we introduced the routing and layout of Blazor. These are the components that we can use to build the navigation hierarchy of our app. By the end of this chapter, we can now perform basic navigation as in the XAML version of our app.

When we built the UIs in this chapter, we saw that the UI design technique of Blazor is the same as web UI design. We can reuse the code from an existing framework, such as Bootstrap.

If we want to create the UI on our own, we can work on the initial design in a playground first. After we are satisfied with the UI design, we can copy HTML and CSS code into our Razor file to build a Razor component. There are many playgrounds used by frontend developers, such as CodePen, JSFiddle, CodeSandbox, and StackBlitz.

In this chapter, we reused Bootstrap examples to build our UIs. Even though this is a straightforward way to implement a web UI, there are many duplicated codes created. In the next chapter, we will refine our code and convert the code into reusable Razor components. We will implement the CRUD operations to add, edit, and delete items using these Razor components.

9

Implementing Blazor Components

In the last chapter, we learned about Blazor routing and layout. We then built a navigation framework by creating the routing and layout of our app. After we created the navigation framework, we created the top-level pages. With the implementation of Razor pages, we can explore the password database in a similar way that we can explore in the XAML version. Razor pages are Razor components, but they are not reusable. Razor components are building blocks of the Blazor UI. In this chapter, we will introduce Razor components. To understand Razor components, we will introduce data binding and the Razor component life cycle. After learning about these concepts, we will refine our code and convert duplicated code into reusable Razor components. Finally, we will use the Razor components that we build to implement CRUD operations in our app.

We will cover the following topics in this chapter:

- Introducing Razor components
- Data binding
- Understanding the life cycle of Razor components
- Implementing CRUD operations

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code of this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter09>

The source code can be downloaded using the following `git` command:

```
git clone -b chapter09 https://github.com/PacktPublishing/.  
NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

Understanding Razor components

Even though we have created and used Razor components in the last two chapters, we haven't taken a closer look at Razor components yet. In this section, we will continue improving the app from the last chapter and dig deeper into Razor components to learn some key concepts about these components.

Blazor apps are built using Razor components. The first Razor component in our app is `Main` and it is defined in `Main.razor`, as shown here:

```
<Router AppAssembly="@typeof(Main).Assembly">  
  <Found Context="routeData">  
    <RouteView RouteData="@routeData"  
      DefaultLayout="@typeof(MainLayout)" />  
  <FocusOnNavigate RouteData="@routeData"  
    Selector="h1" />  
  </Found>  
  <NotFound>  
    <LayoutView Layout="@typeof(MainLayout)">  
      <p role="alert">  
        Sorry, there's nothing at this address.  
      </p>  
    </LayoutView>  
  </NotFound>  
</Router>
```

The `Router` component is installed in the `Main` component, and it handles the routing of pages and selects the default layout component. All other Razor pages are loaded by `Router` components. The Razor pages loaded by `Router` have route templates defined and are used to present the UI to the users. In our project, Razor pages are located in the `Pages` folder. There are also reusable Razor components, and they are the building blocks of Razor pages. These Razor components are in the `Shared` folder.

Basically, each file with the `.razor` file extension is a Razor component and it is compiled into a C# class when it is executed. The class name is the filename. The folder name is used as part of the namespace. For example, the `Login` Razor component is in the `Pages` folder so the folder name, `Pages`, is used as part of the namespace. So, the full name of the `Login` class is `PassXYZ.Vault.Pages.Login`.

We use Pascal case for the class name in C#, so the folder name and Razor filename should use Pascal case as well.

What are Pascal case, camel case, and snake case?

Pascal case, camel case, and snake case are commonly used naming conventions in programming languages. Here are some examples:

- **Camel case** uses uppercase and lowercase in the variable name. The first letter is lowercase, such as `loginUser`.
- **Pascal case** also uses uppercase and lowercase in the variable name, but the first letter is uppercase, such as `LoginUser`.
- **Snake case** uses lowercase only and separates each word with an underscore, such as `login_user`.

Razor components can be authored in a single file or they can be split into a Razor file (`.razor`) and a code-behind C# file (`.cs`). In the code-behind C# file, a partial class is defined to contain all the programming logic. We did this when we created the `Login` component in *Chapter 7, Introducing Blazor Hybrid App Development*.

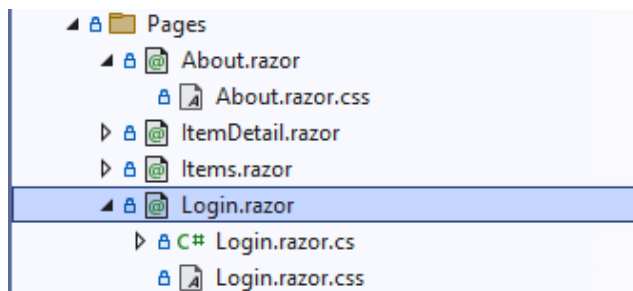


Figure 9.1: Razor component naming convention

When we created the `Login` component, we used Bootstrap CSS style for the styling. Razor components can support CSS isolation, which can simplify CSS and avoid collisions with other components or libraries. Additionally, it can include its own CSS style in a `.razor.css` file.

Inheritance

Since a Razor component is a C# class, it includes all features of a C# class. A Razor component can be a child class of another Razor component. In *Chapter 8, Understanding Blazor Layout and Routing*, when we created layout components, we could see that all layout components are derived classes of `LayoutComponentBase`. As we can see in `MainLayout.razor` in the following code, we use the `@inherits` directive to specify the `LayoutComponentBase` base class:

```
@inherits LayoutComponentBase
<div class="page">
  <div class="sidebar"><NavMenu/></div>
  <main>@Body</main>
</div>
```

All Razor components derive from the `ComponentBase` class, so it is possible to create a Razor component derived from the `ComponentBase` class using the C# file without the Razor markup file. For example, we can create a Razor component called `AppName` in a C# class, as shown here:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Rendering;
namespace PassXYZ.Vault.Pages;
public class AppName : ComponentBase
{
    protected override void BuildRenderTree
        (RenderTreeBuilder builder)
    {
        base.BuildRenderTree(builder);
        builder.OpenElement(0, "div");
        builder.AddContent(1, "PassXYZ.Vault");
        builder.CloseElement();
    }
}
```

`AppName` is a Razor component created without a Razor markup file (`.razor`), but it is the same as other Razor components, as shown here:

```
...
<AppName/>
...
```

We introduced Razor components in this section. We will learn how to package Razor components in a library in the next section.

Creating a Razor class library

In our project, we create reusable components in the `Shared` folder. These components can be reused by other components, such as layout components or `NavMenu`.

We can also encapsulate Razor components in a separate library in the form of the **Razor class library**. The components in the Razor class library are not project-specific, so they can be used in any Blazor project. We can use them in Blazor Hybrid, Blazor WebAssembly, or Blazor Server apps.

In this book, we build Razor components using Bootstrap. There are many open source Razor class libraries built on top of Bootstrap in GitHub. Some of them are good enough for commercial product development. Here are some examples:

- **BootstrapBlazor** – <https://github.com/dotnetcore/BootstrapBlazor>
- **Blazorise** – <https://github.com/Megabit/Blazorise>
- **Havit.Blazor** – <https://github.com/havit/Havit.Blazor/>

These open source projects are built as Razor class libraries so that they can be reused similarly to other .NET libraries. Razor class libraries can be published as **NuGet** packages so we can import them into our Blazor projects.

In this section, we will create a Razor class library similar to the previously mentioned open source projects. We will put Razor components that can be reused in our Razor class library. This library can be published as a NuGet package.

We can create a Razor class library using Visual Studio or the dotnet command line.

To create a Razor class library using Visual Studio, we can add a new project to our solution, as shown in *Figure 9.2*, by following these steps:

1. Search for and select **Razor Class Library** from the project templates.
2. Click **Next** and name the project `PassXYZ.BlazorUI`.
3. On the next screen, click **Create** to create it.

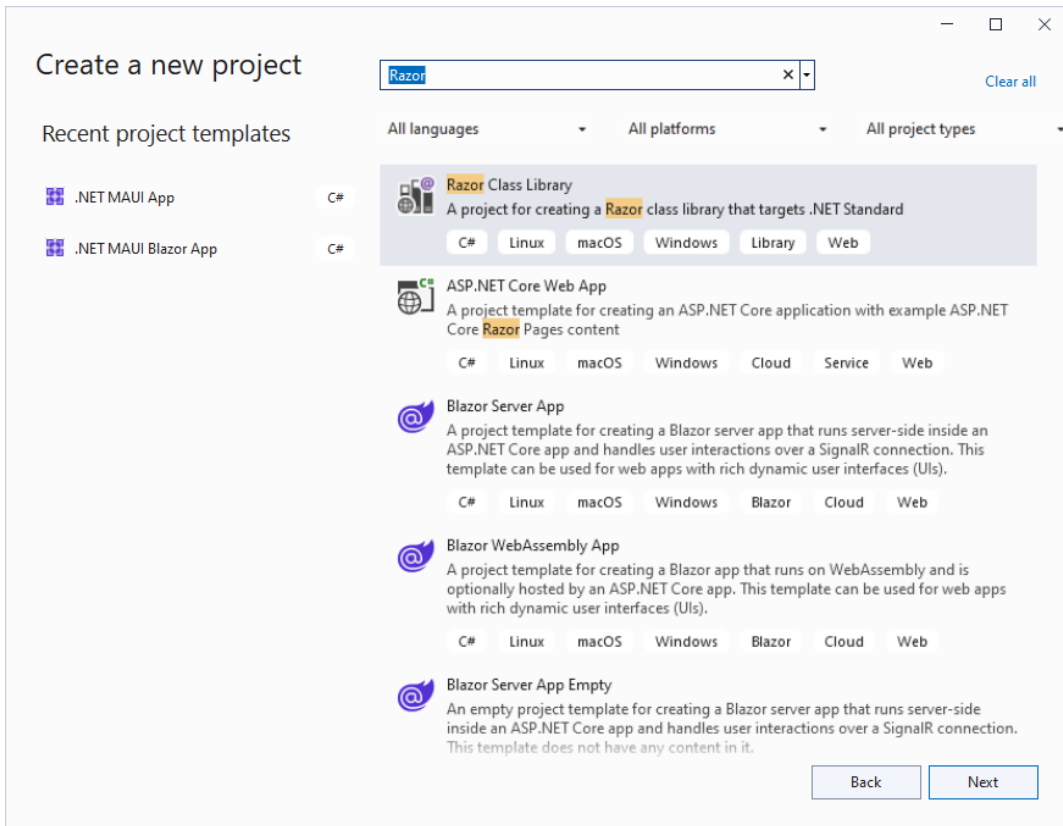


Figure 9.2: Creating a Razor class library

To create the project using a dotnet command line, we can change the directory to the solution folder and execute the following command in Command Prompt:

```
dotnet new razorclasslib -n PassXYZ.BlazorUI
```

The `dotnet new` command will create a new project using the `razorclasslib` template and name the project `PassXYZ.BlazorUI`. To add the project to the solution, we can use the following command:

```
dotnet sln add PassXYZ.BlazorUI\PassXYZ.BlazorUI.csproj
```

We need to remove the unused `Component1.*` and `ExampleJsInterop.cs` files from the `PassXYZ.BlazorUI` project.

To use Razor components in our project, we need to add the project reference into the `PassXYZ.Vault` project. We can add it in Visual Studio by right-clicking the project node and selecting **Add -> Project Reference**. We can also edit the `PassXYZ.Vault.csproj` project file to add the following line:

```
<ItemGroup>
  <ProjectReference
    Include="..\PassXYZ.BlazorUI\PassXYZ.BlazorUI.csproj"
    />
</ItemGroup>
```

To use this library, we need to update the `PassXYZ.Vault\_Imports.razor` file to add the following line:

```
@using PassXYZ.BlazorUI
```

Using static assets in the Razor class library

We use Bootstrap in our Razor components, so we need to include Bootstrap CSS and JavaScript files in the Razor class library. From the Blazor app point of view, we can put these static assets in either the project's `wwwroot` folder or the component library's `wwwroot` folder. Using the Bootstrap CSS file as an example, if we put it in the `wwwroot` project, we can refer to it in `index.html` with the following path:

```
<script src="css/bootstrap/bootstrap.bundle.min.js"/>
```

If we choose to put it in the component library's `wwwroot` folder, we can refer to it with the following path:

```
<script src="_content/PassXYZ.BlazorUI/css/bootstrap/
  bootstrap.bundle.min.js"/>
```

The difference is that we need to refer to the URL in the component library starting with `_content/{LibraryProjectName}`.

After we have created a Razor class library project, we can add more components to it.

Creating reusable Razor components

In this section, we can create reusable components by optimizing our code. Throughout the process, we can get a better understanding of the features of Razor components and how to make them reusable.

We created the Blazor Hybrid version of our app, *PassXYZ.Vault* in *Chapter 7, Introducing Blazor Hybrid App Development*, and we added layout and routing functionalities to it in *Chapter 8, Understanding*

the *Blazor Layout and Routing*. Our app can browse and update the password database now. So far, we haven't implemented most of the CRUD operations. We will add these functionalities after we refine our Razor components in this chapter.

To navigate the password database, we created two Razor components – `Items` and `ItemDetail` – in *Chapter 7, Introducing .NET MAUI Blazor*. The `Items` class is used to display a list of password entries and groups in the current group, and the `ItemDetail` class is used to display the content of a password entry.

If we look at the layout of `Items` and `ItemDetail` as shown in *Figure 9.3*, the look and feel of both pages are quite similar:

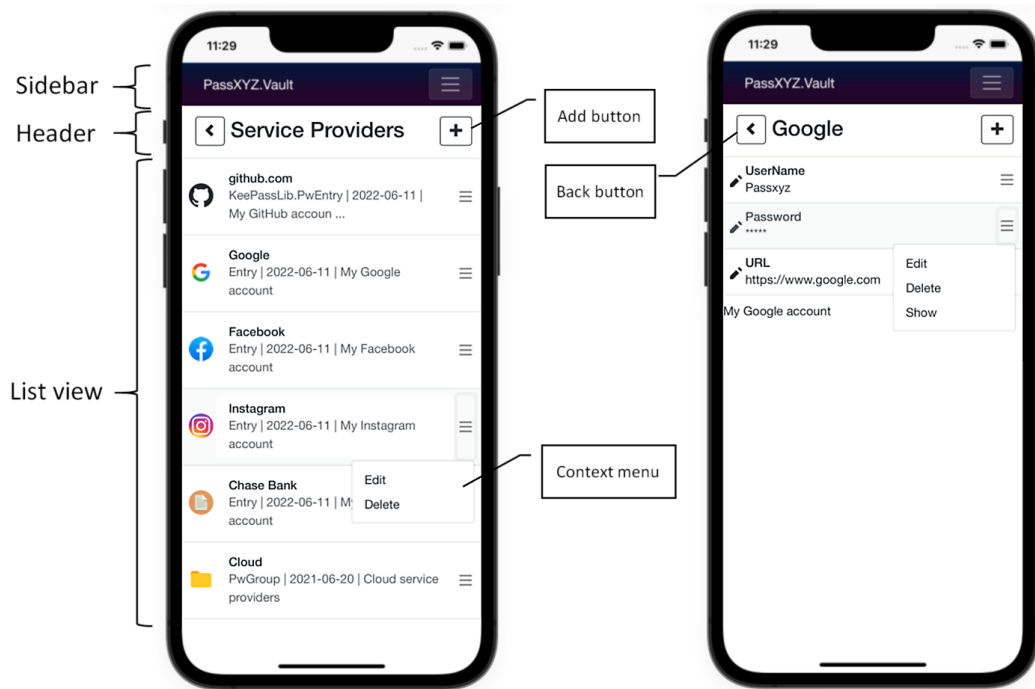


Figure 9.3: UI layout of `Items` and `ItemDetail`

The layout of both pages includes a sidebar, a header, and a list view. The sidebar is defined in the `Layout` component. The header and list view are implemented in both `Items` and `ItemDetail` with partially duplicated code there. We will optimize our code and abstract the duplicated code into reusable components in this chapter and the next chapter.

There are two buttons, **Add** and **Back**, in the header. The **Back** button can be used to navigate back to the parent group, and the **Add** button can be used to add a new item or field.

In the list view item, we can use the context menu to perform item-level operations, such as edit or delete. The context menu includes menu items to perform specific actions related to the selected item or field. For the edit or delete CRUD operations, after the menu item is selected, a modal dialog related to the action is displayed.

In the current implementation, both `Items` and `ItemDetail` include all UI elements in one Razor markup. We will start to refine the code into smaller reusable components to make our implementation clean.

We will convert modal dialogs into Razor components in this chapter and convert the header and list view into Razor components in the next chapter. Let us start with modal dialogs. To support add, edit, and delete operations, we need two kinds of dialog boxes:

- Editor dialog – adding or editing items or fields
- Confirmation dialog – to confirm before deleting an item or a field

In *Chapter 8, Understanding the Blazor Layout and Routing*, we used the HTML and CSS code from Bootstrap examples to implement modal dialogs. We haven't investigated them in detail yet, since our markup files look long and complex. We will analyze the code and turn it into Razor components in this chapter.

Creating a base modal dialog component

To refine Editor and Confirmation dialogs, we can build a base modal dialog first. Using this base modal dialog, we can create either Editor or Confirmation dialogs.

To create a new Razor component in the `PassXYZ.BlazorUI` project, we can right-mouse-click on the project node and select **Add -> New Item... -> Razor Component** in the project template. We name the Razor component as `ModalDialog` and create a C# code-behind file for it. After that, we type the code in *Listing 9.1* to `ModalDialog.razor` and *Listing 9.2* to `ModalDialog.razor.cs`.

The UI code is extracted from the `Items` or `ItemDetail` code in *Chapter 8, Understanding the Blazor Layout and Routing*, as shown in *Listing 9.1*:

Listing 9.1: `ModalDialog.razor` (<https://epa.ms/ModalDialog9-1>)

```
<div class="modal fade" id=@Id tabindex="-1"
    aria-labelledby="ModelLabel" aria-hidden="true">
  <div class="modal-dialog"><div class="modal-content">
    <div class="modal-header"> ❶
      <h5 class="modal-title" id="ModelLabel">@Title</h5> ❷
      <button type="button" class="btn-close" ❸
        data-bs-dismiss="modal" aria-label="Close"/>
```

```

</div>
<div class="modal-body">                                ❷
  <form class="row gx-2 gy-3">
    @ChildContent                                       ❸
    <div class="col-12">
      <button type="button" class="btn btn-secondary"
        data-bs-dismiss="modal" @onclick=
          "OnClickClose">
        @CloseButtonText                               ❹
      </button>
      <button type="submit" class="btn btn-primary"
        data-bs-dismiss="modal" @onclick=
          "OnClickSave">
        @SaveButtonText                               ❺
      </button>
    </div>
  </form>
</div>
</div></div>
</div>

```

From the markup code in *Listing 9.1*, we can see that this is a typical HTML code snippet in Bootstrap style. We embedded C# variables in HTML to create the component UI.

This base dialog UI includes a header ❶ and body ❷. There is a title ❸ and a close button ❹ in the header. Inside the body, we can see a child content area ❺ and two buttons (Close ❹/Save ❺). We can refer to *Figure 9.4* to see the layout of this base modal dialog:

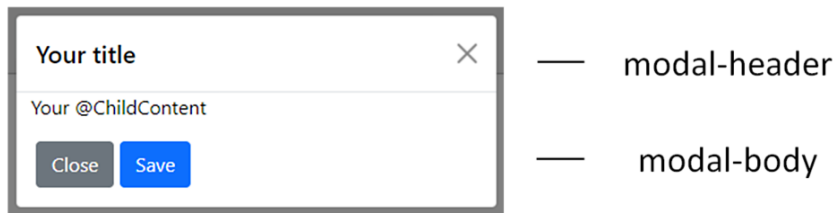


Figure 9.4: Base dialog

Even though the HTML and CSS code is very similar to the Bootstrap example, we replaced all the hardcode content with C# variables. If we use this modal dialog component to build a new component, the following is an example:

```
<ModalDialog Id=@id Title="Please confirm" OnSaveAsync=
    @OnDelete
    SaveButtonText="Save" CloseButtonText="Close">
    Do you want to delete UserName?
</ModalDialog>

<button class="dropdown-item" data-bs-toggle="modal"
    data-bs-target="#@Id">Please confirm</button>
```

In the preceding markup code, we define the modal dialog using the `<ModalDialog>` component tag. Each modal dialog has a unique ID to identify it. We can show the dialog box after clicking a button. In the button, we provide the modal dialog ID to identify it.

Inside the `<ModalDialog>` component tag, we assigned the value of multiple attributes defined in the `ModalDialog` component, such as the ID, title, text of buttons, event handler, and so on.

Data binding

Instead of assigning a string or data directly to the attribute of an HTML element, we can assign a variable to it. This is the data binding feature of the Razor component. We will learn how to use data binding in this section.

In data binding, when we assign a variable to the attribute of the DOM element, the data flows from Razor components to DOM elements. When we respond to the DOM event, the data flows from DOM elements to Razor components. Since we can use a Razor component just like a DOM element, the data flow between child and parent Razor components is similar to the data exchange between Razor components and DOM elements.

For example, we can bind the `id` variable to the `Id` attribute of `ModalDialog` and we can handle the button click event using the `OnDelete` event handler:

```
<ModalDialog Id=@id Title="Please confirm" OnSaveAsync=
    @OnDelete
    SaveButtonText="Save" CloseButtonText="Close">
```

In the preceding example, the data flows from the `id` variable to the `Id` attribute of `ModalDialog`. When the `OnDelete` event handler is invoked, the data flows from `ModalDialog` back to the current context. The `ModalDialog` attributes, `Id` and `OnSaveAsync`, are defined in the C# code-behind file. Let's review the C# code-behind file of `ModalDialog` in the next section.

Component parameters

We can define the attributes of Razor components using component parameters. To define component parameters, we must create public properties with the `[Parameter]` attribute.

In the `ModalDialog` class, as shown in *Listing 9.2*, we declare seven component parameters, `Id`, `Title`, `ChildContent`, `OnClose`, `OnSaveAsync`, `CloseButtonText`, and `SaveButtonText`. We can use these component parameters in data binding:

Listing 9.2: `ModalDialog.razor.cs` (<https://epa.ms/ModalDialog9-2>)

```
using Microsoft.AspNetCore.Components;
using System.Diagnostics;
using System.Diagnostics.CodeAnalysis;

namespace PassXYZ.BlazorUI;

public partial class ModalDialog : IDisposable
{
    [Parameter]
    public string? Id { get; set; } ❶

    [Parameter]
    public string? Title { get; set; } ❷

    [Parameter]
    public RenderFragment ChildContent { get; set; } ❸

    [Parameter]
    public Func<Task>? OnClose { get; set; } ❹

    [Parameter]
    public Func<Task<bool>>? OnSaveAsync { get; set; } ❺

    [Parameter]
    [NotNull]
    public string? CloseButtonText { get; set; } ❻

    [Parameter]
    [NotNull]
    public string? SaveButtonText { get; set; } ❼

    private async Task OnClickClose() {
        if (OnClose != null) { await OnClose(); }
    }
}
```

```
private async Task OnClickSave() {
    if (OnSaveAsync != null) { await OnSaveAsync(); }
}
void IDisposable.Dispose() {
    GC.SuppressFinalize(this);
}
}
```

The component parameters of `ModalDialog` are defined as follows:

- **Id ❶** – This is used to identify a modal dialog
- **Title ❷** – This is the title of the modal dialog
- **ChildContent ❸** – This is where the content of the child component should be inserted

Two event handlers – **OnClose ❹** and **OnSaveAsync ❺** – are defined to handle button click actions. We can customize the text of both buttons using **CloseButtonText ❻** and **SaveButtonText ❼**.

We can treat component parameters just like HTML attributes. We can assign a C# field, property, or return value of a method to the component parameter of `ModalDialog`.

After we create the `ModalDialog` base component, we can create `Editor` and `Confirmation` dialog components using it.

Let's create a new modal dialog, `ConfirmDialog`, to ask for the confirmation of deleting an item. To create a new `ConfirmDialog` component in the `PassXYZ.BlazorUI` project, we can right-mouse-click on the project node and select **Add -> New Item... -> Razor Component** in the project template. We can name the Razor component `ConfirmDialog` and type the following code as shown in *Listing 9.3*:

Listing 9.3: `ConfirmDialog.razor` (<https://epa.ms/ConfirmDialog9-3>)

```
<ModalDialog Id=@Id Title=@($w"Deleting {Title}") OnSaveAsync=
@OnSavew
    SaveButtonText="Confirm" CloseButtonText="Cancel">
    Please confirm to delete @Title?
</ModalDialog>

@code {w
    [Parameter]
    public string Id { get; set; } = "confirmDialog"w; ❶
    [Parameter]
```

```

public string? Title { get; set; }                                ②
[Parameter]
public Action? OnConfirmClick { get; set; }

async Task<bool> OnSave() {
    OnConfirmClick?.Invoke();
    return true;
}
}

```

We define the `Id` ① and `Title` ② component parameters in `ConfirmDialog` and pass their values to the base class through data binding. We also subscribe to the `OnSaveAsync` event using the `OnSave` event handler. We also define our own event handler, `OnConfirmClick`, as a component parameter to which other components can subscribe.

In `ConfirmDialog`, we actually bind parameters through nested components. In this case, the data should flow in the directions suggested here:

- Change notifications flow up the hierarchy
- New parameter values flow down the hierarchy

The values of the `Id` and `Title` attributes are assigned by the components that use `ConfirmDialog`, and their values flow down to `ModalDialog`. The `Save` or `Close` button events are triggered in the `ModalDialog` component, and they flow up the chain to `ConfirmDialog` and upper-level components. If we use the `Save` button as an example, the event flows up in the direction shown here:

```
onclick (DOM) -> OnSaveAsync (ModalDialog) -> OnConfirmClick (ConfirmDialog)
```

It starts from the `onclick` event in DOM. `ModalDialog` defines its own event, `OnSaveAsync`, which is triggered by the `onclick` event handler. `ConfirmDialog` defines its own event, `OnConfirmClick`, which is triggered by the `OnSaveAsync` event handler.

Nested components

`ConfirmDialog` is one of the examples of nested components. As we can see, we can embed components inside components by declaring them using HTML syntax. The embedded components look like HTML tags, where the name of the tag is the component type. For example, we can use `ModalDialog` inside `ConfirmDialog`, as shown here:

```
<ModalDialog ...>Please confirm to delete @Title?</ModalDialog>
```

Nested components are the way to build the component hierarchy in Blazor. Inheritance and composition are the two ways that we can extend and reuse a class in an object-oriented programming language. In Blazor, composition is used in nested components to extend the functionalities. Inheritance is an *is-a* relationship, while composition is a *has-a* relationship. In nested components, the parent component has a child component in it.

In Microsoft Blazor and ASP.NET Core documents, the terms *ancestor* and *descendant* or *parent* and *child* are used to explain the relationship of nested components. Here, parent and child is not an inheritance relationship, but a composition relationship. A better term could be an outer component or an inner component. Nevertheless, to be consistent with the Microsoft documentation, I won't choose a different term in the discussion. Please just be aware that when we discuss nested components and data binding, the ancestor and descendant relationship is a *has-a* relationship or composition.

In our previous example, the `ConfirmDialog` component is the outer component, while `ModalDialog` is the inner component. The relationship is that `ConfirmDialog`, has `ModalDialog` in it.

Child content rendering

When we build nested components, there are many cases in which one component can set the content of another component. The outer component provides the content between the inner component's opening and closing tags. In `ConfirmDialog`, it sets the content of `ModalDialog` as follows:

```
<ModalDialog Id=@Id Title=@($"Deleting {Title}")
  OnSaveAsync=@OnSave
  SaveButtonText="Confirm" CloseButtonText="Cancel">
  Please confirm to delete @Title?
</ModalDialog>
```

This is done by using a special component parameter called `ChildContent`, which is of the `RenderFragment` type. In the preceding code, the `Please confirm to delete @Title?` string is set to the `ChildContent` parameter of `ModalDialog`.

`ConfirmDialog` is still a relatively simple example of nested components. Let's look at another example, `EditorDialog`, to explore more Razor component features. As we mentioned earlier, we need two dialog boxes to handle add, edit, and delete actions. `ConfirmDialog` is used to confirm with users before deleting an item or a field. To add or edit an item or a field, we need a dialog box that can provide editing features.

We can do the same to create the new component, `EditorDialog`. After selecting **Add -> New Item...** -> **Razor Component** in the project template, we can name the Razor component `EditorDialog` and create a C# code-behind file for it. After that, we type the code in *Listing 9.4* to `EditorDialog.razor` and *Listing 9.5* to `EditorDialog.razor.cs`.

Let's review the Razor markup code of `EditorDialog` as shown in *Listing 9.4*:

Listing 9.4: EditorDialog.razor (<https://epa.ms/EditorDialog9-4>)

```
<ModalDialog Id=@Id Title=@Key OnSaveAsync=@OnSaveClicked
  SaveButtonText = "Save" CloseButtonText="Close">
  @if (IsKeyEditingEnable) { ❶
    <input type="text" class="form-control" id="keyField"
      @bind="Key" placeholder=@KeyPlaceHolder required> ❷
    }
  @ChildContent
  <div>
    <textarea class="form-control" id="valueField"
      style="height: 100px"
      placeholder=@ValuePlaceHolder
      @bind="Value" required /> ❸
  </div>
</ModalDialog>
```

`EditorDialog` is built using `ModalDialog`. It can be used to edit a key-value pair. When we create a new key-value pair, we want to edit both the key and the value. When we edit an existing key-value pair, we may want to make a change to the value field only. These are the two use cases that we want to support in `EditorDialog`. The condition is detected using a component parameter called `IsKeyEditingEnable` **❶**. The key portion of the UI is rendered as an `<input>` **❷** element when we want to create a new key-value pair. When we edit an existing key-value pair, the key is displayed as the title in the header area, and we edit the value in the `<textarea>` **❸** element. This is the main functionality of our `EditorDialog` component.

We can see the UI in *Figure 9.5*. On the left-hand side, it shows the dialog when we want to add a new field. We need to provide the field name and content. On the right-hand side, it shows the dialog when we want to edit an existing URL field. The field name is displayed in the title, and we can change the content in `<textarea>`:

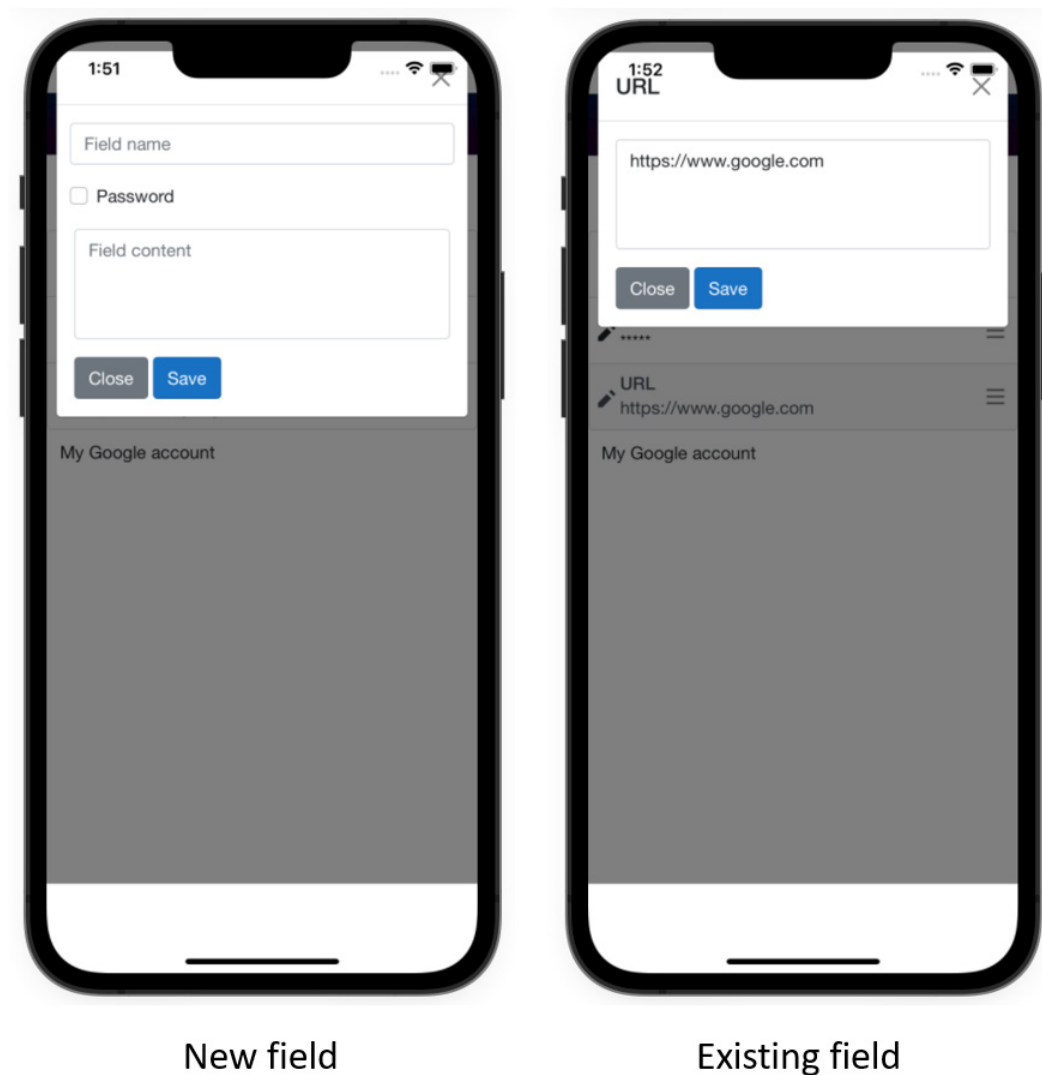


Figure 9.5: Editing a field

In the `EditorDialog` component, when we edit the key and value using the `<input>` and `<textarea>` HTML elements, the initial value is displayed. The initial value sets from the Razor component to the DOM. After we make the changes, the data flows from the DOM to the Razor component. This is two-way data binding.

Two-way data binding

Two-way data binding can be created with the `@bind` Razor directive attribute. With this syntax, an HTML element attribute can bind to a field, property, expression value, or result of a method. In *Listing 9.4*, the `<input>` element value binds to the `Key` property in the `EditorDialog` component:

```
<input type="text" class="form-control" id="keyField"
      @bind="Key" placeholder="@KeyPlaceholder" required>
```

With two-way data binding, the DOM element `<input>` value is updated whenever the `Key` property is changed. The `Key` property is updated as well when the user updates the `<input>` value in the DOM.

In the preceding example, instead of using the `@bind` directive attribute, we can replace the `@bind` directive attribute with two one-way data bindings, as you can see in the following code:

```
<input type="text" class="form-control" id="keyField"
      value="@Key"
      @onchange="@((ChangeEventArgs e) => Key = e?.Value?
                  .ToString())"
      placeholder="@KeyPlaceholder" required>
```

When our `EditorDialog` component is rendered, the value of the `<input>` element comes from the `Key` property. When the user enters a value in the textbox and changes the element focus, the `onchange` event is fired and the `Key` property is set to the changed value.

For the `<input>` element, the default event of the `@bind` directive attribute is the `onchange` event. We can change the event with an `@bind:event="{event}"` attribute. The `{event}` placeholder should be a DOM event. For example, we can change the `onchange` event to an `oninput` event with the following code snippet:

```
<input type="text" class="form-control" id="keyField"
      @bind="Key" @bind:event="oninput" placeholder="@KeyPlaceholder"
      required>
```

Binding with component parameters

In the previous section, we discussed two-way data binding between a Razor component and a DOM element. Since the Razor component can be used in a similar way as the DOM element, we can create a two-way data binding between two Razor components as well. This is usually the case when we need to communicate between parent and child (inner or outer) components.

We can bind a component parameter of an inner component to the property of an outer component with the `@bind-{PROPERTY}` syntax. The `{PROPERTY}` placeholder is the property to bind. We explained that the `@bind` directive attribute can be replaced by two one-way data binding setups,

which include assigning a variable to the `<input>` value attribute and assigning an event handler to the `onchange` event. The event handler can be added automatically for `@bind` by the compiler, but not for `@bind-{PROPERTY}`. We need to define our own event of the `EventCallback<TValue>` type to bind with component parameters. The event name must be `{PARAMETER NAME}Changed`. Let's use our `EditorDialog` component to explain how to use the `@bind-{PROPERTY}` directive attribute.

In our code, we edit a field using `EditorDialog` in the `ItemDetail` component or edit an item using the same in the `Items` component. Let's use field editing as an example:

```
<EditorDialog Id=@_dialogEditId
  @bind-Key="listGroupField.Key"                                ❶
  @bind-Value="listGroupField.Value"                            ❷
  IsKeyEditingEnable=@_isNewField OnSave="UpdateFieldAsync"
  KeyPlaceHolder="Field name" ValuePlaceHolder="Field
    content">
  @if (_isNewField) {
    <div class="form-check">
      <input class="form-check-input" type="checkbox"
        @bind="listGroupField.IsProtected"
        id="flexCheckDefault">
      <label class="form-check-label"
        for="flexCheckDefault">
        Password
      </label>
    </div>
  }
</EditorDialog>
```

In the preceding code of the `ItemDetail` component, we can create data binding of Key ❶ and Value ❷ to the `listGroupField` of the `Field` type. We need to implement the `{PARAMETER NAME}Changed` events in C# code-behind of `EditorDialog`, as shown here in *Listing 9.5*:

Listing 9.5: `EditorDialog.razor.cs` (<https://epa.ms/EditorDialog9-5>)

```
namespace PassXYZ.BlazorUI;

public partial class EditorDialog {
  [Parameter]
```

```

public string? Id { get; set; }
bool _isKeyEditingEnable = false;
[Parameter]
public bool IsKeyEditingEnable ...
[Parameter]
public EventCallback<bool>? IsKeyEditingEnableChanged {
    get; set; }

string _key = string.Empty;
[Parameter]
public string Key {
    get => _key;
    set {
        if(_key != value) {
            _key = value;
            KeyChanged?.InvokeAsync(_key);
        }
    }
}
[Parameter]
public EventCallback<string>? KeyChanged { get; set; }
[Parameter]
public string? KeyPlaceholder { get; set; }
string _value = string.Empty;
[Parameter]
public string Value ...
[Parameter]
public EventCallback<string>? ValueChanged { get; set; }
[Parameter]
public string? ValuePlaceholder { get; set; }
[Parameter]
public RenderFragment ChildContent { get; set; } =
    default!;
[Parameter]
public Action<string, string>? OnSave { get; set; }
async Task<bool> OnSaveClicked() {

```

```

        OnSave?.Invoke(Key, Value);
        return true;
    }
}

```

In *Listing 9.5*, we use the `Key` property as an example to explain the process of component parameter binding. The `Key` property is defined as a component parameter with the `[Parameter]` attribute. An associated event is defined as `KeyChanged` of the `EventCallback<TValue>` type. When the user changes the text input and changes the element focus, the setter of the `Key` property is invoked. Inside the setter of the `Key` property, it fires the `KeyChanged` event, which will inform the outer `ItemDetail` component. As a result, the `listGroupField.Key` linked variable is updated.

Communicating with cascading values and parameters

We can use data binding to pass data between parent and child components. Data binding is good to pass data to the intermediate child component. Sometimes, we may want to pass data to components several levels deep. If we use data binding in this situation, then we have to create multiple levels of chained data binding. The complexity increases with the chained levels. For example, if we want to pass data from `Items` to `ModalDialog`, we have to create a data binding to `ConfirmDialog` first. Then, another level of data binding needs to be created between `ConfirmDialog` and `ModalDialog`.

In `Items`, we need to pass the `Id` dialog to `ModalDialog`. We need to use an `Id` dialog to identify the dialog instance that we want to display. As we can see next, we define `ConfirmDialog` in the `Items` component. `Id` is defined in `Items` and passes to `ConfirmDialog` using the component parameter:

```

<ConfirmDialog Id="@_dialogDeleteId" Title=
    @listGroupItem.Name
    OnConfirmClick="DeleteItemAsync" />

```

Then, `ConfirmDialog` has to pass it to `ModalDialog`:

```

<ModalDialog Id=@Id Title=@($"Deleting {Title}")
    OnSaveAsync=@OnSave
    SaveButtonText="Confirm" CloseButtonText="Cancel">
    Please confirm to delete @Title?
</ModalDialog>

```

In `ModalDialog`, `Id` is used as an attribute of the `<div>` element:

```

<div class="modal fade" id=@Id tabindex="-1"
    aria-labelledby="ModelLabel" aria-hidden="true"> ...

```

To avoid multiple levels of data binding, we can use cascading values and parameters as a method to flow data down a component hierarchy.

`CascadingValue` is a component of the Blazor framework. The outer component provides a cascading value using `CascadingValue`, and the inner component can receive it using the `[CascadingParameter]` attribute. To demonstrate the usage, we can change the code of the `Items` component as follows:

```
<CascadingValue Value="@_dialogDeleteId" Name="Id">
  <ConfirmDialog Title=@listGroupItem.Name
    OnConfirmClick="DeleteItemAsync" />
</CascadingValue>
```

We use cascading value with the `<CascadingValue>` tag. In the `<CascadingValue>` tag, we pass the `_dialogDeleteId` variable to the `Value` attribute and the `"Id"` string to the `Name` attribute. Since this `Id` is not used by `ConfirmDialog` directly, the `Id` component parameter can be removed from `ConfirmDialog`.

In `ModalDialog`, we change the `Id` property from a component parameter to a parameter using the `[CascadingParameter]` attribute:

```
[CascadingParameter (Name = "Id")]
public string Id { get; set; } = default!;
```

If we have only one cascading value, we don't have to specify the cascading value name. The compiler can help us to find it by data type. However, to avoid ambiguities, we can name the cascading value using the `Name` attribute. Let's look at the final changes in the `Items` component using the cascading value for both `ConfirmDialog` and `EditorDialog`:

```
<CascadingValue Value="@_dialogEditId" Name="Id">
  <EditorDialog @bind-Key="listGroupItem.Name"
    @bind-Value="listGroupItem.Notes"
    IsKeyEditingEnable=true
    OnSave="UpdateItemAsync" KeyPlaceholder="Item name"
    ValuePlaceholder="Please provide a description">
    @if (_isNewItem) {
      <select @bind="newItem.SubType" class="form-select"
        aria-label="Group">
        <option selected value=@ItemSubType.Group>
          @ItemSubType.Group</option>
        <option value=@ItemSubType.Entry>
```

```
        @ItemSubType.Entry</option>
        <option value=@ItemSubType.PxEntry>
            @ItemSubType.PxEntry</option>
        <option value=@ItemSubType.Notes>
            @ItemSubType.Notes</option>
    </select>
}
</EditorDialog>
</CascadingValue>

<CascadingValue Value="@_dialogDeleteId" Name="Id">
    <ConfirmDialog Title=@listGroupItem.Name
        OnConfirmClick="DeleteItemAsync" />
</CascadingValue>
```

As we can see, after we use a cascading value, `ConfirmDialog` and `EditorDialog` don't need to handle the `Id` field directly. The code is more concise than the previous version.

In this section, we discussed how to create reusable components. Some Razor components may have dependencies on data or network services. We need to take extra actions during the creation or the destruction of the components. We can do these as part of the life cycle management of Razor components.

Let us review the life cycle of Razor components in the next section.

Understanding the component lifecycle

A Razor component has a lifecycle just like any other object. There is a set of synchronous and asynchronous lifecycle methods that can be overridden to help developers perform additional operations during component initialization and rendering.

We can review the Razor component lifecycle in *Figure 9.6*:

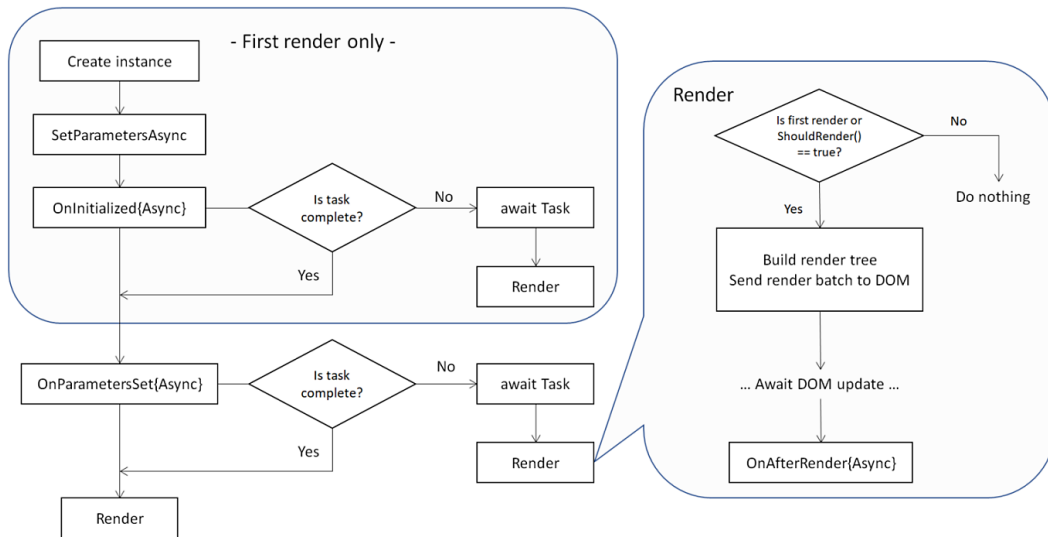


Figure 9.6: Razor component lifecycle

In *Figure 9.6*, we can see that we can add hooks during the initialization and rendering phases. The following methods can be overridden to catch initialization events:

- `SetParametersAsync`
- `OnInitialized` and `OnInitializedAsync`
- `OnParametersSet` and `OnParametersSetAsync`

`SetParametersAsync` and `OnInitialized(Async)` are invoked only in the first render. `OnParametersSet(Async)` is called every time a parameter is changed.

The following methods can be overridden to customize rendering:

- `ShouldRender`
- `OnAfterRender` and `OnAfterRenderAsync`

We will review these lifecycle methods in detail and show how we use them in our code.

SetParametersAsync

`SetParametersAsync` is the first hook after the object is created and it has the following signature:

```
public override Task SetParametersAsync(ParameterView parameters)
```

The `ParameterView` parameter contains component parameters or cascading parameter values. `SetParametersAsync` sets the value of each property with the `[Parameter]` or `[CascadingParameter]` attribute. This function can be overridden to add logic that needs to be executed before the parameters are set. The next hook after `SetParametersAsync` is `OnInitialized{Async}`.

OnInitialized and OnInitializedAsync

`OnInitialized` and `OnInitializedAsync` are invoked when the component is initialized. They have the following signatures, respectively:

```
protected override void OnInitialized()  
protected override async Task OnInitializedAsync()
```

By overriding these two functions, we can add logic to initialize our component here. Please be aware that they are only called once, right after the creation of the component. For time-consuming initialization tasks, the asynchronous method can be used, such as downloading data using RESTful API calls. As we can see in *Figure 9.6*, after an asynchronous method is completed, the DOM needs to be rendered again.

OnParametersSet and OnParametersSetAsync

When component parameters are set or changed, `OnParametersSet` and `OnParametersSetAsync` are invoked. We can see that there are two versions to handle both synchronous and asynchronous cases. The asynchronous version of `OnParametersSetAsync` can be used to handle time-consuming tasks. Once the asynchronous task is completed, the DOM needs to be rendered again to reflect any changes.

The methods have the following signatures, respectively:

```
protected override void OnParametersSet()  
protected override async Task OnParametersSetAsync()
```

These two methods will be invoked whenever component parameters or cascading parameters are changed. They can be called multiple times, while `OnInitialized{Async}` is only called once.

As we can see in *Figure 9.6*, the DOM can be rendered multiple times during the initialization phase due to which asynchronous calls may be invoked. The methods involved in the rendering process are `ShouldRender` and `OnAfterRender{Async}`.

ShouldRender

The `ShouldRender` method returns a Boolean value, indicating whether the component should be rendered. As we can see in *Figure 9.6*, the first render ignores this method, so a component should be rendered at least once. This method has the following signature:

```
protected override bool ShouldRender()
```

OnAfterRender and OnAfterRenderAsync

`OnAfterRender` and `OnAfterRenderAsync` are called after a component has finished rendering. They have the following signatures, respectively:

```
protected override void OnAfterRender(bool firstRender)
protected override async Task OnAfterRenderAsync(bool
    firstRender)
```

They can be used to perform additional initialization tasks with the rendered content, such as invoking JavaScript code in the component. This method has a Boolean `firstRender` parameter, which allows us to attach JavaScript event handlers only once. There is an asynchronous version of this method, but the framework won't schedule a further render cycle after the asynchronous task is completed.

To have a look at the effect of lifecycle methods, we can run a test to add all lifecycle methods in the `ConfirmDialog` component, as you can see here:

```
public ConfirmDialog()
{
    Debug.WriteLine($"ConfirmDialog-{Id}: is created");
}
public override Task SetParametersAsync
    (ParameterView parameters)
{
    Debug.WriteLine($"ConfirmDialog-{Id}:
        SetParametersAsync called");
    return base.SetParametersAsync(parameters);
}
protected override void OnInitialized()
    => Debug.WriteLine($"ConfirmDialog-{Id}: OnInitialized
        called - {Title}");
protected override async Task OnInitializedAsync() =>
    await Task.Run(() => {
```

```

    Debug.WriteLine($"ConfirmDialog-{Id}: OnInitializedAsync
        called - {Title}");
});
protected override void OnParametersSet()
    => Debug.WriteLine($"ConfirmDialog-{Id}: OnParametersSet
        called - {Title}");
protected override async Task OnParametersSetAsync() =>
    await Task.Run(() => {
Debug.WriteLine($"ConfirmDialog-{Id}:
    OnParametersSetAsync called - {Title}");
});
protected override void OnAfterRender(bool firstRender)
    => Debug.WriteLine($"ConfirmDialog-{Id}: OnAfterRender
        called with firstRender = {firstRender}");
protected override async Task OnAfterRenderAsync(bool
    firstRender) => await Task.Run(() => {
    Debug.WriteLine($"ConfirmDialog-{Id}:
        OnAfterRenderAsync called - {Title}");
});
protected override bool ShouldRender() {
    Debug.WriteLine($"ConfirmDialog-{Id}: ShouldRender called
        - {Title}");
    return true;
}

```

We override all lifecycle methods in `ConfirmDialog` and add debug output to show the progress. After we launch our app, we can see the following output:

```

ConfirmDialog-: is created
ConfirmDialog-: SetParametersAsync called
ConfirmDialog-deleteModel: OnInitialized called -
ConfirmDialog-deleteModel: OnInitializedAsync called -
ConfirmDialog-deleteModel: OnParametersSet called -
ConfirmDialog-deleteModel: OnParametersSetAsync called -
ConfirmDialog-deleteModel: ShouldRender called -
ConfirmDialog-deleteModel: ShouldRender called -
ConfirmDialog-deleteModel: OnAfterRender called with

```

```
firstRender = True
ConfirmDialog-deleteModel: OnAfterRenderAsync called -
ConfirmDialog-deleteModel: OnAfterRender called with
firstRender = False
ConfirmDialog-deleteModel: OnAfterRenderAsync called -
ConfirmDialog-deleteModel: OnAfterRender called with
firstRender = False
ConfirmDialog-deleteModel: OnAfterRenderAsync called -
```

The preceding output is the one when we just launch our app and the Items page is shown. We can see that the `Id` cascading parameter is not set before the `SetParametersAsync` method is called. Since we override the asynchronous methods, there are multiple render cycles scheduled in parallel. `ShouldRender` and `OnAfterRender{Async}` are invoked multiple times due to rendering occurring in parallel.

Let's look at another case when we click on the context menu on the Items page. When we click on the context menu of an item, such as Google, `ConfirmDialog` is initialized again. The output is as follows:

```
ConfirmDialog-deleteModel: SetParametersAsync called
ConfirmDialog-deleteModel: OnParametersSet called - Google
ConfirmDialog-deleteModel: ShouldRender called - Google
ConfirmDialog-deleteModel: OnParametersSetAsync called -
Google
ConfirmDialog-deleteModel: ShouldRender called - Google
ConfirmDialog-deleteModel: OnAfterRender called with
firstRender = False
ConfirmDialog-deleteModel: OnAfterRenderAsync called -
Google
ConfirmDialog-deleteModel: OnAfterRender called with
firstRender = False
ConfirmDialog-deleteModel: OnAfterRenderAsync called -
Google
```

The `SetParametersAsync` method is called again since the `Title` component parameter is changed. We can see that the `Title` component parameter is set to `Google` in the subsequent calls.

In our code, we use `OnParametersSet` to load the list of items in `Items.razor.cs` and load a list of `Field` in `ItemDetail.razor.cs`. Let's review `OnParametersSet` in `ItemDetail.razor.cs`:

```
protected override void OnParametersSet() {
    base.OnParametersSet();

    if (SelectedItemId == null) {
        throw new InvalidOperationException(
            "ItemDetail: SelectedItemId is null");
    }

    selectedItem = DataStore.GetItem(SelectedItemId, true);
    if (selectedItem == null) {
        throw new InvalidOperationException(
            "ItemDetail: entry cannot be found with SelectedItemId");
    }
    else {
        if (selectedItem.IsGroup) {
            throw new InvalidOperationException(
                "ItemDetail: SelectedItemId should not be a group
                here.");
        }

        fields.Clear();
        List<Field> tmpFields = selectedItem.GetFields();
        foreach (Field field in tmpFields) {
            fields.Add(field);
        }
        notes = selectedItem.GetNotesInHtml();
    }
}
```

❶ In `OnParametersSet`, we check whether the `SelectedItemId` component parameter is null. This is the ID of the selected item. ❷ If it is not null, we can find the item by calling the `IDataStore` method called `GetItem()`. ❸ Once we get the instance of the selected item, we can get a list of fields by calling the `GetFields()` method.

In `Items.razor.cs`, the implementation of `OnParametersSet` is very similar to this. You can refer to the following GitHub link to find out the details:

<https://epa.ms/Items9-6>

So far, we have an almost full-function password manager app, and the UI of this app is built with Blazor. We created reusable modal dialog components to support the context menu so we can perform CRUD operations. The last piece of the puzzle is the implementation of CRUD operations.

Implementing CRUD operations

Once we have prepared modal dialogs, which will be used in CRUD operations from previous sections, we can implement CRUD operations in this section.

CRUD operations of items

To add or update an item, we can use the `UpdateItemAsync()` method in `Items.razor.cs` to handle both cases. To detect whether we want to create a new item or update an existing item, we define a private `_isNewItem` field as follows:

```
bool _isNewItem = false;
```

Next, we'll see how to add or edit an item.

Adding a new item

To add a new item, we can click the `+` button in the header of the `Items` page, as shown in *Figure 9.7*:

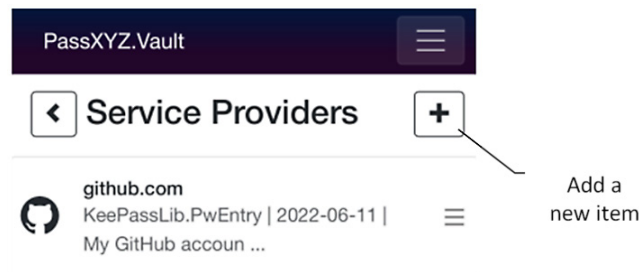


Figure 9.7: Adding a new item

The Razor markup of this page header can be reviewed here:

```
<div class="container"><div class="row">
  <div class="col-12"><h1>
    @if (selectedItem?.GetParentLink() != null) {
```

```

    <a class="btn btn-outline-dark"
      href="@selectedItem?.GetParentLink()">
      <span class="oi oi-chevron-left"
        aria-hidden="true"></span></a>
    }
    @(" " + Title)
  <button type="button"
    class="btn btn-outline-dark float-end"
    data-bs-toggle="modal"
    data-bs-target="#@_dialogEditId"
    @onclick="@(() => _isNewItem=true)">
    <span class="oi oi-plus" aria-hidden="true">
  </span></button>
</h1></div>
</div></div>

```

The page header displays the **Back** button ❶, Title ❷, and the **Add** button ❸. The **Back** button is displayed if the parent link exists.

When the **Add** button is clicked, it will display a modal dialog with Id defined in the `_dialogEditId` variable. The `onclick` event handler just sets `_isNewItem` to `true` so the modal dialog event handler knows this is an action to add a new item.

Editing or deleting an item

To edit or delete an item, we can click on the context menu on the item, as shown in *Figure 9.8*:

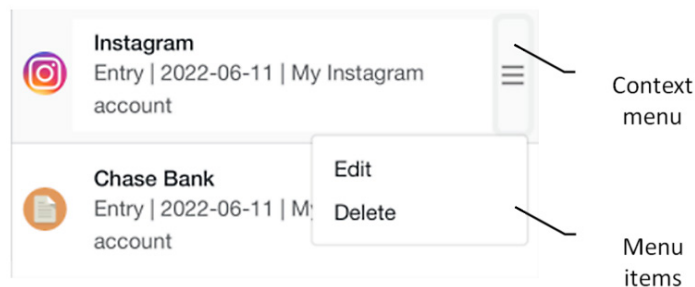


Figure 9.8: Editing or deleting an item

After we click on the context menu button, a list of menu items will be displayed. Let's review the markup for the context menu in `Items.razor` as follows:

```
<div class="list-group">
  @foreach (var item in items) {
<div class="dropdown list-group-item list-group-item-action
  d-flex gap-1 py-2" aria-current="true">
  
  <a href="@item.GetActionLink()" class="..."> ...
  <button class="opacity-50 btn btn-light
    dropdown-toggle" type="button"
    id="itemsContextMenu"
    data-bs-toggle="dropdown" aria-expanded="false"
    @onclick="@(() => listGroupItem=item)"> ❶
    <span class="oi oi-menu" aria-hidden="true"></span>
  </button>
  <ul class="dropdown-menu" aria-labelledby=
    "itemsContextMenu">
    <li><button class="dropdown-item"
      data-bs-toggle="modal"
      data-bs-target="#@_dialogEditId"
      @onclick="@(() => _isNewItem=false)"> ❷
      Edit</button></li>
    <li><button class="dropdown-item"
      data-bs-toggle="modal"
      data-bs-target="#@_dialogDeleteId"> ❸
      Delete</button></li>
  </ul>
  </div>
  }
</div>
```

There is a context menu button ❶ defined in the preceding markup code. When this button is clicked, two menu items, Edit ❷ and Delete ❸, will be displayed. Since the markup code of the context menu runs in a `foreach` loop, we need to get a reference of the selected item to edit or delete it. In the logic of C# code-behind, the `listGroupItem` variable is used to refer to the selected item. We can catch the reference in the `onclick` event handler of the context menu button.

When the Edit menu item is selected, we need to set the `_isNewItem` variable to `false` so the event handler of the modal dialog can know we are editing an existing item.

With all the previous setup, let's review the event handler in modal dialogs. Let's review the `UpdateItemAsync()` event handler in `Items.razor.cs` first:

```
private async void UpdateItemAsync(string key, string value) {
    if (listGroupItem == null) { return; }
    if (string.IsNullOrEmpty(key) || string.IsNullOrEmpty(
        value))
    { return; }

    listGroupItem.Name = key;
    listGroupItem.Notes = value;
    if (_isNewItem) {                                ❶
        // Add new item
        if (listGroupItem isNewItem aNewItem) {
            Item? newItem = DataStore.CreateNewItem
                (aNewItem.SubType);
            if (newItem != null) {
                newItem.Name = aNewItem.Name;
                newItem.Notes = aNewItem.Notes;
                items.Add(newItem);
                await DataStore.AddItemAsync(newItem);
            }
        }
    }
    else {
        // Update the current item
        await DataStore.UpdateItemAsync(listGroupItem);
    }
}
```

The `UpdateItemAsync()` event handler can handle both adding and editing an item. As we can see, it checks the `_isNewItem` variable ① to detect whether we want to add or edit an item. After that, it calls `IDataStore` methods to process add or update actions.

Next, let's review the event handler of deleting an item:

```
private async void DeleteItemAsync() {
    if (listGroupItem == null) return;

    if (items.Remove(listGroupItem)) {
        _ = await DataStore.DeleteItemAsync
            (listGroupItem.Id);
    }
}
```

In the `DeleteItemAsync()` event handler, it just removes the item from the list and calls `IDataStore` methods to process the delete action.

CRUD operations of fields

The CRUD operations of fields are similar to what we have done for items. To add or update a field, we can use the `UpdateFieldAsync()` method in `ItemDetail.razor.cs` to handle both cases. To detect whether we want to create a new field or update an existing field, we define a private `_isNewField` field as follows:

```
bool _isNewField = false;
```

The UI of CRUD operations is also similar to what we have explained in the previous section. Please refer to *Figure 9.9* to see the **Add** button and context menu items:

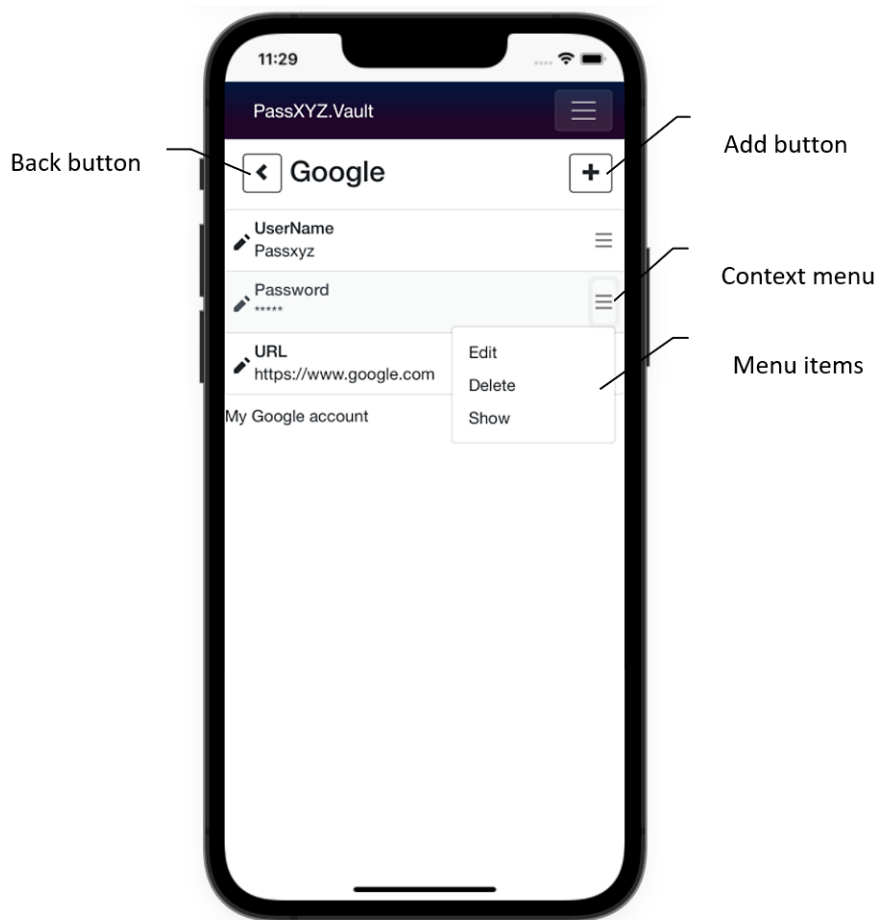


Figure 9.9: Add, edit, or delete a field

We can review the Razor markup code of the page header in `IwwtemDetail.razor` as follows:

```
<div class="container">
  <div class="row"><div class="col-12">
    <h1>
      @if (selectedItem?.GetParentLink() != null) {
        <a class="btn btn-outline-dark"
          href="@selectedItem?.GetParentLink()">
          <span class="oi oi-chevron-left"
            aria-hidden="true"></span></a>
```

```

    }
    @" " + selectedItem!.Name)
    <button type="button" class="btn btn-outline-dark
        float-end"
        data-bs-toggle="modal" data-bs-
            target="#@_dialogEditId"
        @onclick="@(() => _isNewField=true)">
        <span class="oi oi-plus"
            aria-hidden="true"></span></button>
    </h1>
</div></div>
</div>

```

As we can see, the preceding source code is also similar to the one in `Items.razor` except the `_isNewItem` variable is replaced by `_isNewField`. We can refine this page header to a reusable component later.

Just like in the previous section, let's review the source code of the list group and context menu:

```

<div class="list-group">
    @foreach (var field in fields) {
        @if (field.ShowContextAction == null) {
            <div class="dropdown list-group-item ...
                aria-current="true">
                <span class="oi oi-pencil" aria-hidden="true">
                    </span>
                <div class="d-flex gap-2 w-100
                    justify-content-between"> ...
                <button class="opacity-50 btn btn-light
                    dropdown-toggle" type="button"
                    id="itemDetailContextMenu"
                    data-bs-toggle="dropdown" aria-expanded="false"
                    @onclick="@(() => listGroupField=field)"> ❶
                    <span class="oi oi-menu" aria-hidden="true">
                        </span>
                    </button>
                <ul class="dropdown-menu"
                    aria-labelledby="itemDetailContextMenu">

```

```

        <li><button class="dropdown-item"
            data-bs-toggle="modal"
            data-bs-target="#@_dialogEditId"
            @onclick="@(() => _isNewField=false)"> ❷
            Edit
        </button></li>
        <li><button class="dropdown-item"
            data-bs-toggle="modal"
            data-bs-target="#@_dialogDeleteId"> ❸
            Delete
        </button></li>
        @if (field.IsProtected) {
            <li><button class="dropdown-item"
                @onclick="OnToggleShowPassword"> ❹
                @if (field.IsHide) { <span>Show</span> }
                else { <span>Hide</span> }
            </button></li>
        }
    </ul>
</div>
}
}
</div>

```

The preceding source code of `ItemDetail.razor` includes a context menu button ❶ and three buttons for the **Add** ❷, **Edit** ❸, and **Show** ❹ menu items. You can see that the source code is also similar to the one in `Items.razor`, which includes a list group and a context menu. We will refine this to a reusable component in the next chapter. The difference in the context menu is we add a menu item to show or hide the field if the field is a protected field, such as a password. We use the `onclick` event handler, `OnToggleShowPassword()`, to set the `IsHide` field property to toggle the visibility of the password field.

Finally, let's review the event handlers of modal dialogs in `ItemDetail.razor.cs`:

```

private async void UpdateFieldAsync(string key, string
    value) {
    if (selectedItem == null || listGroupField == null) {
        throw new NullReferenceException("Selected item is
            null");
    }
}

```

```

    }
    if (string.IsNullOrEmpty(key) ||
        string.IsNullOrEmpty(value)) { return; }
    listGroupField.Key = key;
    listGroupField.Value = value;

    if (_isNewField) {
        // Add a new field
        Field newField =
            selectedItem.AddField(listGroupField.Key,
                ((listGroupField.IsProtected) ?
                    listGroupField.EditValue :
                    listGroupField.Value),
                    listGroupField.IsProtected);
        fields.Add(newField);
    }
    else {
        // Update the current field
        var newData = (listGroupField.IsProtected) ?
            listGroupField.EditValue : listGroupField.Value;
        selectedItem.UpdateField(listGroupField.Key,
            newData, listGroupField.IsProtected);
    }
    await DataStore.UpdateItemAsync(selectedItem);
}

```

The `UpdateFieldAsync()` event handler handles both adding and editing a field. It is called with two parameters – key and value. The corresponding arguments are passed from the modal dialog and we use them to set the field of `listGroupField`. The handler checks the `_isNewField` variable to detect whether we want to add or edit a field. After that, it calls `IDataStore` methods to process add or update actions.

To remove a field, the following `DeleteFieldAsync()` event handler is invoked:

```

private async void DeleteFieldAsync() {
    if (listGroupField == null || selectedItem == null) {
        throw new NullReferenceException(

```

```
        "Selected item or field is null");  
    }  
    listGroupField.ShowContextAction = listGroupField;  
    selectedItem.DeleteField(listGroupField);  
    await DataStore.UpdateItemAsync(selectedItem);  
}
```

In the `DeleteFieldAsync()` event handler, we just delete the field from the selected item and call the `IDataStore` method to update the database.

With the implementation of CRUD operations, we have concluded this section. Now, we have a new version of the password manager app using Blazor UI. The difference between this version and the one in *Part 1* of this book is that we use Blazor to build all UIs. The look and feel of Blazor UI are similar to web apps, while XAML UI is the same as native apps.

Summary

In this chapter, we introduced how to create Razor components. We learned about data binding and the component lifecycle. After that, we created a set of modal dialog components to clean up our code. With Razor components, we can remove duplicated code and improve the UI design. We implemented CRUD operations in the event handlers of modal dialogs. We now have a new version of the password manager app.

During the code analysis, we can see that we still have redundant code in the two main components, `Items` and `ItemDetail`. Even though we optimized modal dialogs, we still have duplicated code in the list group and context menu. We will convert them to Razor components in the next chapter.

Advanced Topics in Creating Razor Components

In Blazor app development, everything is a component. We learned how to create Razor components in the last chapter. In this chapter, we will explore more advanced topics about Razor components. To convert list groups and context menus to Razor components, we need to understand advanced topics such as templated components and form validation. We will introduce these concepts while we are creating more Razor components in our project.

We will cover the following topics in this chapter:

- Creating more Razor components
- Using templated components
- Built-in Razor components and input validation

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code of this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter10>

The source code can be downloaded using the following Git command:

```
git clone -b chapter10 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

Creating more Razor components

We developed Modal dialog components in *Chapter 9, Razor Components and Data Binding*. In this chapter, we will refine our code to remove the duplicated code in `Items` and `ItemDetail` pages and convert the duplicated code into Razor components. We will create the following components:

- `Navbar` – This is a component to display a navigation bar
- `Dropdown` – This is a component to support the context menu
- `ListView` – This is a component to display a list of items

The `ListView` component is the most complicated one so we will leave it till the end of this section. Let's work on `Navbar` and `Dropdown` first.

Creating the Navbar component

Let's look at the navigation bar UI in *Figure 10.1*. We can see that the navigation bar contains a **Back** button, a title, and an **Add** button:

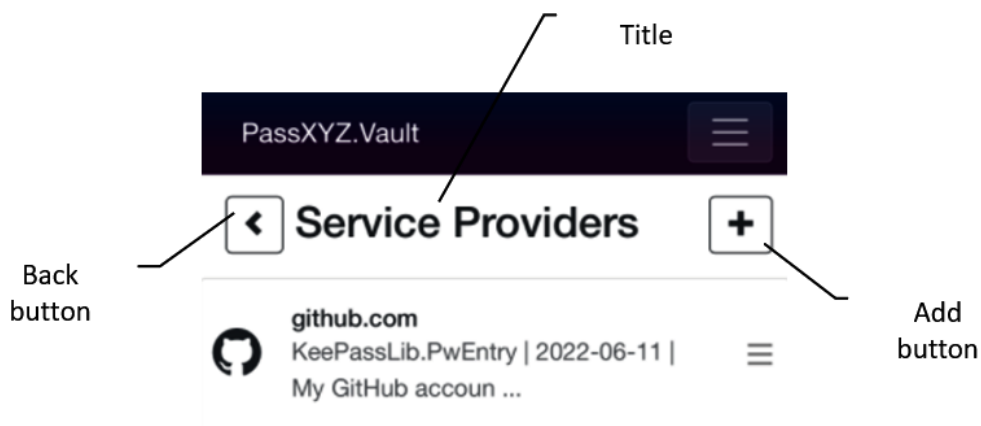


Figure 10.1: Navigation bar

The current code of the navigation bar is shown next, and this code snippet is duplicated on both the `Items` and `ItemDetail` pages:

```
<div class="container">
  <div class="row">
    <div class="col-12">
      <h1>
```

```

        @if (selectedItem?.GetParentLink() != null) { ❶
            <a class="btn btn-outline-dark"
                href="@selectedItem?.GetParentLink()">
                <span class="oi oi-chevron-left"
                    aria-hidden="true"></span></a> ❷
        }
        @(" " + Title) ❸
        <button type="button"
            class="btn btn-outline-dark float-end"
            data-bs-toggle="modal"
            data-bs-target="#@_dialogEditId"
            @onclick="@(() => _isNewItem=true)"> ❹
            <span class="oi oi-plus" aria-hidden="true">
            </span>
        </button>
    </h1>
</div>
</div>
</div>

```

In the preceding code, **❶** the **Back** button is shown when there is a parent link. **❷** The **Back** button is implemented as a `<a>` tag. **❸** `Title` is a string and is shown as part of the `<h1>` tag. **❹** The **Add** button is implemented as a `<button>` tag. Bootstrap style is used to format both the **Back** and **Add** buttons.

To convert the preceding code to a Razor component, we can create a new Razor component in the `PassXYZ.BlazorUI` project and name it `Navbar`. `Navbar` can display the UI elements in *Figure 10.1*, which include a **Back** button, a title, and an **Add** button. To separate the UI and logic, we create both a `Navbar.razor.cs` C# code-behind file and Razor markup, `Navbar.razor`. We define component parameters and event handlers in the C# code-behind file, as shown in *Listing 10.1*:

Listing 10.1: `Navbar.razor.cs` (<https://epa.ms/Navbar10-1>)

```

public partial class Navbar
{
    [Parameter]
    public string? ParentLink { get; set; } ❶
    [Parameter]

```

```

public string? DialogId { get; set; }           ❷
[Parameter]
public string? Title { get; set; }             ❸
[Parameter]
public EventCallback<MouseEventArgs> OnAddClick { get;
set; }                                         ❹
private void OnClickClose(MouseEventArgs e) {
    OnAddClick.InvokeAsync();
}
}

```

There are four component parameters and an event handler defined in `Navbar`. We can set the parent link of the **Back** button with the `ParentLink` parameter ❶. The value of `Title` is set to the `Title` parameter ❸. For the **Add** button, we need to provide an ID and an event handler for the dialog box so the `DialogId` ❷ and `OnAddClick` ❹ parameters are used.

Now, let us look at the Razor file of `Navbar` in *Listing 10.2*:

Listing 10.2: `Navbar.razor` (<https://epa.ms/Navbar10-2>)

```

@namespace PassXYZ.BlazorUI

<div class="container">
    <div class="row">
        <div class="col-12">
            <h1>
                @if (ParentLink != null) {           ❶
                    <a class="btn btn-outline-dark"
                        href="@ParentLink">           ❶
                        <span class="oi oi-chevron-left"
                            aria-hidden="true"></span>
                    </a>
                }
                @(" " + Title)                       ❸
            <button type="button"
                class="btn btn-outline-dark float-end"

```

```

        data-bs-toggle="modal"
        data-bs-target="#@DialogId"
        @onclick="OnClickClose">
        <span class="oi oi-plus" aria-hidden="true">
        </span>
    </button>
</h1>
</div>
</div>
</div>

```

We can see that the code is very similar to the one in `Items` and `ItemDetail`. The difference is that we replaced the hardcode value with component parameters (`ParentLink` ①, `DialogId` ②, `Title` ③, and `OnClickClose` ④). With this new `Navbar` component, we can replace the code in `Items` using the `Navbar` component as follows:

```

<Navbar ParentLink="@selectedItem?.GetParentLink()"
        Title="@Title" DialogId="@_dialogEditId"
        OnAddClick="@(() => {_isNewItem=true;})" />

```

And we can replace the code in `ItemDetail` as follows:

```

<Navbar ParentLink="@selectedItem?.GetParentLink()"
        Title="@selectedItem?.Name" DialogId="@_dialogEditId"
        OnAddClick="@(() => {_isNewField=true;})" />

```

As we can see, we refined the code by removing duplicated code, and the new code looks much more elegant and concise.

We have done the work for `Navbar`. Now let's move to the `Dropdown` component.

Creating a Dropdown component for the context menu

To create a component similar to the context menu, we can reuse the Bootstrap `Dropdown` component. As we can see in *Figure 10.2*, a context menu includes a context menu button and a list of menu items. When users click on the context menu button, a list of menu items is displayed:

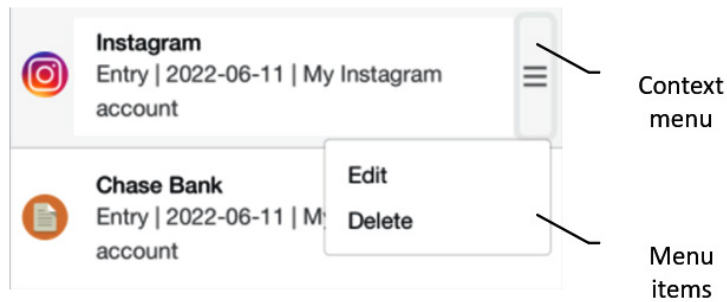


Figure 10.2: Context menu

The current code of the context menu is duplicated on both the Items and ItemDetail pages, which are shown here:

```
<button class="opacity-50 btn btn-light dropdown-toggle"
  type="button" id="itemsContextMenu"
  data-bs-toggle="dropdown"
  aria-expanded="false"
  @onclick="@(() => listGroupItem=item)">
  <span class="oi oi-menu" aria-hidden="true"></span>
</button>
<ul class="dropdown-menu"
  aria-labelledby="itemsContextMenu">
  <li><button class="dropdown-item" data-bs-toggle="modal"
    data-bs-target="#@_dialogEditId"
    @onclick="@(() => _isNewItem=false)">
    Edit
  </button></li>
  <li><button class="dropdown-item" data-bs-toggle="modal"
    data-bs-target="#@_dialogDeleteId">
    Delete
  </button></li>
</ul>
```

The Dropdown component of Bootstrap includes a button and an unordered list. We need to define an event handler for the button to take action. In the preceding code, we set the `item` variable to `listGroupItem`. For the menu items, each menu item is implemented as a `<button>` tag and it accepts a dialog ID and an event handler as parameters. When a menu item is clicked, the corresponding modal dialog will be shown.

We can create two new Razor components in the `PassXYZ.BlazorUI` project and name them as `Dropdown` and `MenuItem`. We can also implement them in the C# code-behind file (*Listing 10.4*) and Razor file (*Listing 10.3*) to separate the UI and logic, which we'll be doing now.

Let's review the Dropdown component UI first in *Listing 10.3*:

Listing 10.3: Dropdown.razor (<https://epa.ms/Dropdown10-3>)

```
@namespace PassXYZ.BlazorUI
<button class="opacity-50 btn btn-light dropdown-toggle"
    type="button" id="itemDetailContextMenu"
    data-bs-toggle="dropdown"
    aria-expanded="false" @onclick="OnClick">
    <span class="oi oi-menu" aria-hidden="true"></span>
</button>
<ul class="dropdown-menu"
    aria-labelledby="itemDetailContextMenu">
    @ChildContent
</ul>
```

In the Dropdown component, we define a button ❶ and an unordered list ❷. The click event of the button is defined as an `OnClick` event handler. The items in the unordered list are displayed as child content of the Dropdown component. The component parameters are defined in the C# `Dropdown.razor.cs` code-behind file in *Listing 10.4*:

Listing 10.4: Dropdown.razor.cs (<https://epa.ms/Dropdown10-4>)

```
namespace PassXYZ.BlazorUI;

public partial class Dropdown
{
    [Parameter]
    public EventCallback<MouseEventArgs> OnClick {get;set;}❶
    [Parameter]
```

```
public RenderFragment ChildContent { get; set; }      ②
}
```

In `Dropdown.razor.cs`, two component parameters – `OnClick` ① and `ChildContent` ② – are defined.

The `MenuItem` component can be displayed as the child content of the `Dropdown` component. We can see the UI code of `MenuItem` in *Listing 10.5*:

Listing 10.5: MenuItem.razor (https://epa.ms/MenuItem10-5)

```
@namespace PassXYZ.BlazorUI

<li>
  <button class="dropdown-item" data-bs-toggle="modal"
    data-bs-target="#@Id" @onclick="OnClick">
    @ChildContent
  </button>
</li>
```

The `MenuItem` component defines three component parameters – `Id`, `OnClick`, and `ChildContent`. These parameters are defined in `MenuItem.razor.cs` in *Listing 10.6*:

Listing 10.6: MenuItem.razor.cs (https://epa.ms/MenuItem10-6)

```
namespace PassXYZ.BlazorUI;

public partial class MenuItem
{
  [Parameter]
  public string? Id { get; set; }      ①
  [Parameter]
  public EventCallback<MouseEventArgs> OnClick {get; set;} ②
  [Parameter]
  public RenderFragment ChildContent { get; set; }      ③
}
```

① The `Id` parameter is used to specify the dialog ID when the menu item is clicked. ② `OnClick` is used to register an event handler for a button click event. ③ `ChildContent` is used to display child content such as the menu item name.

We have implemented the components for the context menu. We can replace the redundant code in the `Items` and `ItemDetail` pages with context menu components. On the `Items` page, the context menu is implemented as follows:

```
<Dropdown OnClick="@(() => currentItem.
Data=listGroupItem=item)">
  <MenuItem Id="@_dialogEditId"
    OnClick="@(() => _isNewItem=false)">Edit</MenuItem>
  <MenuItem Id="@_dialogDeleteId">Delete</MenuItem>
</Dropdown>
```

On the `ItemDetail` page, the context menu is implemented as follows:

```
<Dropdown OnClick="@(() =>
{currentField.Data=listGroupField=field;})">
  <MenuItem Id="@_dialogEditId"
    OnClick="@(() => _isNewField=false)">Edit</MenuItem>
  <MenuItem Id="@_dialogDeleteId">Delete</MenuItem>
  @if (field.IsProtected) {
  <MenuItem OnClick="OnToggleShowPassword">
    @(field.IsHide ? "Show":"Hide")
  </MenuItem>
  }
</Dropdown>
```

After we refined the code of the `Items` and `ItemDetail` pages, we created a modal dialog, navigation bar, and context menu components. The code looks much more elegant and concise now. However, we still have room to refine the code further. The main UI logic in both the `Items` and `ItemDetail` pages is a list view. We can refine this part of the code as a `ListView` component. To create a `ListView` component, we need to use an advanced feature called templated components.

Using templated components

To build a Razor component, component parameters are the channels for parent and child communication. In *Chapter 9, Razor Components and Data Binding*, we introduced nested components. We mentioned a special `ChildContent` component parameter of the `RenderFragment` type. The parent component can set the content of the child component with this parameter. For example, the content of `MenuItem` in the following code can be set to an HTML string:

```
<MenuItem Id="@_dialogDeleteId">
  <strong>Delete</strong>
</MenuItem>
```

We can do this because `MenuItem` defines the following component parameter as we can see in *Listing 10.6*:

```
[Parameter]
public RenderFragment ChildContent { get; set; }
```

If we want to explicitly specify the `ChildContent` parameter, we can do this as well:

```
<MenuItem Id="@_dialogDeleteId">
  <ChildContent>
    <strong>Delete</strong>
  </ChildContent>
</MenuItem>
```

`ChildContent` is a special component parameter that we can implicitly use in the markup language. To use `ChildContent`, we define a component that can accept a UI template of the `RenderFragment` type as a component parameter. We can define more than one UI template as a parameter when we create a new component. This kind of component is called a templated component.

A render fragment of the `RenderFragment` type represents a segment of the UI to render. There is also a generic version, `RenderFragment<TValue>`, which takes a type parameter. We can specify a type when `RenderFragment` is invoked.

Creating a `ListView` component

To create `ListView`, we need to use multiple UI templates as component parameters. We can create a new Razor component in the `PassXYZ.BlazorUI` project and name it `ListView`. As we did for `Navbar` and the context menu, we separate the UI and code in a Razor file (*Listing 10.7*) and a C# code-behind file (*Listing 10.8*):

Listing 10.7: `ListView.razor` (<https://epa.ms/ListView10-7>)

```
@namespace PassXYZ.BlazorUI
@typeparam TItem

<div class="list-group">
  @if (Header != null) {
    @Header
  }
  @if (Row != null && Items != null) {
    @foreach (var item in Items) {
```

❶

```

        <div class="dropdown list-group-item
            list-group-item-action
            d-flex gap-1 py-2" style="border: none"
            aria-current="true">
            @Row.Invoke(item)
        </div>
    }
}
@if (Footer != null) {
    <div class="container">
        <article>@Footer</article>
    </div>
}
</div>

```

In the `ListView` Razor file, we define three UI templates – Header ❶, Row ❷, and Footer ❸. We render Header and Footer the same as `ChildContent`, but the Row component parameter looks different. We render it as follows:

```
@Row(item)
```

Or we can render it like this:

```
@Row.Invoke(item)
```

We render it with an `item` argument. The type of Row is `RenderFragment<TValue>`, as we can see in *Listing 10.8*:

Listing 10.8: `ListView.razor.cs` (<https://epa.ms/ListView10-8>)

```

namespace PassXYZ.BlazorUI;

public partial class ListView<TItem>
{
    [Parameter]
    public RenderFragment? Header { get; set; }
    [Parameter]
    public RenderFragment<TItem>? Row { get; set; }
    [Parameter]

```

```

public IEnumerable<TItem>? Items { get; set; }           ③
[Parameter]
public RenderFragment? Footer { get; set; }           ④
}

```

We define `ListView` as a generic `ListView<TItem>` type with the `TItem` type parameter. In the `ListView` component, we can define a list view header using the `Header` ① parameter and the footer using `Footer` ④. `ListView` can be bound to any data collection of the `IEnumerable<TItem>` type using the `Items` parameter ③. The `Row` parameter ② can be used to define the UI template for an individual item in the `foreach` loop.

Using the `ListView` component

Now, we can look at the usage of the `ListView` component in the `Items` and `ItemDetail` pages. We use the `ItemDetail` page as an example here:

```

<ListView Items="fields">                               ①
  <Row Context="field">                                   ②
    @if (field.ShowContextAction == null) {
      <span class="oi oi-pencil" aria-hidden="true"></span>
      <div class="d-flex gap-2 w-100
        justify-content-between">
        <div>
          <h6 class="mb-0">@field.Key</h6>
          <p class="mb-0">@field.Value</p>
        </div>
      </div>
    }
  </div>
  <Dropdown
    OnClick="@(() =>
      {currentField.Data=listGroupField=field;})">
    <MenuItem Id="@_dialogEditId"
      OnClick="@(() => _isNewField=false)">
      Edit
    </MenuItem>
    <MenuItem Id="@_dialogDeleteId">Delete</MenuItem>
    @if (field.IsProtected) {
      <MenuItem OnClick="OnToggleShowPassword">

```

```
        @(field.IsHide ? "Show":"Hide")
    </MenuItem>
    }
</Dropdown>
}
</Row>
<Footer>
    @( (MarkupString) notes)
</Footer>
</ListView>
```

Since we define `Header`, `Row`, and `Footer` as optional parameters, we don't have to specify all of them. In the `ItemDetail` page, we use `Row` and `Footer`. ❶ We need to pass the list of fields to the `Items` parameter first. ❷ Each field in the `foreach` loop is passed to `ListView` as an argument to `Row`, which is defined here:

```
<Row Context="field">
```

The `"field"` value of the `Context` property is used to specify the argument to `Row`. Inside the UI template of `Row`, we display the key value of `field` and create a context menu using the `Dropdown` and `MenuItem` components that we implemented in the last section.

Using the `ListView` component, we can see that the implementation of the `ItemDetail` page looks much better now.

We have made the improvement by creating our own Razor components.

In Blazor, there are a few options for us to develop a UI. We can use HTML/CSS for the UI design. On top of HTML/CSS, we can build our own Razor components to improve the design and implementation. There are also many third-party Razor component libraries available for us, and we can also use ASP.NET Core built-in Razor components.

In the next section, we will support data validation using ASP.NET built-in Razor components.

Built-in components and validation

In our app, we implement `EditorDialog` as a key value editor. When we create or edit an item or a field, we use it to edit a pair of key-value data. `EditorDialog` is built using the Bootstrap framework.

One key feature missing in `EditorDialog` is that it doesn't support data validation. Data validation includes two layers – the UI layer and logic. We can implement simple data validation logic in C#, but the UI of data validation is much more complicated. We haven't done it in the UI layer yet. In both

the `Items` and `ItemDetail` pages, we implemented simple data validation logic. We can review the data validation logic in the `UpdateItemAsync()` method of the `Items` page:

```
private async Task<bool> UpdateItemAsync(string key, string
value)
{
    if (listGroupItem == null) return false;
    if (string.IsNullOrEmpty(key) ||
        string.IsNullOrEmpty(value))
        return false;

    listGroupItem.Name = key;
    listGroupItem.Notes = value;

    if (!_isNewItem) {...}
    else {...}
    StateHasChanged();
    return true;
}
```

In `EditorDialog`, after we complete the editing, `UpdateItemAsync()` is invoked to save data. We check the value of the `key` and `value` arguments first before we continue. If any of them is null, we just return false. There is no problem in the program logic here, but we should notify users about the error, which is due to data saving. With ASP.NET built-in components, data validation is supported at the UI layer. The user can get instant feedback about the error.

Using built-in components

We actually used built-in components in previous chapters. When we introduced routing and the layout of Razor components, we used the `Router`, `RouteView`, `LayoutView`, and `MainLayout` components in `Main.razor`. They are all built-in components.

In this section, we will explore built-in input components, which we can use to enhance the experience of editing components with data validation support. The following table is a list of the built-in input components:

Input component	HTML tag
<code>InputCheckbox</code>	<code><input type="checkbox"></code>
<code>InputDate<TValue></code>	<code><input type="date"></code>
<code>InputFile</code>	<code><input type="file"></code>

InputNumber<TValue>	<input type="number">
InputRadio<TValue>	<input type="radio">
InputRadioGroup<TValue>	Group of child InputRadio<TValue>
InputSelect<TValue>	<select>
InputText	<input>
InputTextArea	<textarea>
EditForm	<form>

Table 10.1: Built-in components

To find detailed information about built-in input components, you can check the following Microsoft document:

<https://learn.microsoft.com/en-us/aspnet/core/blazor/forms-and-input-components?view=aspnetcore-6.0>

In *Table 10.1*, we can see a list of input components and an `EditForm` component. The built-in input components are enhanced versions of the corresponding HTML elements listed in the right-hand column. When we use built-in input components with `EditForm`, `EditForm` can coordinate both validation and submission events. The built-in input components can validate user input when a form is submitted.

Using the EditForm component

The `EditForm` component is an enhanced version of the HTML `<form>` element. To use `EditForm`, we can refer to the following code, which shows an empty `EditForm` component:

```
<EditForm Model="ModelData" OnSubmit="HandleSubmit"
  OnInvalidSubmit="HandleInvalidSubmit"
  OnValidSubmit="HandleValidSubmit">
</EditForm>
```

Or, it can be used in another way:

```
<EditForm EditContext="_editContext" OnSubmit="HandleSubmit"
  OnInvalidSubmit="HandleInvalidSubmit"
  OnValidSubmit="HandleValidSubmit">
</EditForm>
```

We can pass data to it using the `Model` parameter or `EditContext`.

We can specify an instance of a class as `Model` to be edited in `EditForm`. An instance of `EditContext` will be created based on the assigned model instance by `EditForm`. `EditContext` is used as a cascading value for other components in the form. We can also specify an `EditContext` instance directly if we want to take control of it.

To handle the result of form editing, we can register the following callback functions:

- `OnInvalidSubmit` – This callback will be invoked when the form is submitted and `EditContext` is invalid
- `OnSubmit` – This callback will be invoked when the form is submitted
- `OnValidSubmit` – This callback will be invoked when the form is submitted and `EditContext` is valid

We will use the `EditForm` component and built-in input components to create a new `EditFormDialog` component to enhance our key value editor with data validation support.

Creating an `EditFormDialog` component

To support data validation, the `EditForm` component needs to be bound to a model that uses data annotations. The built-in input components should be used for the form editing so that enhanced features such as data validation can be used.

First, we need to create a model class, `KeyValueData`, that can be used for key-value editing in the `PassXYZ.BlazorUI` project, as shown in *Listing 10.9*:

Listing 10.9: `KeyValueData.cs` (<https://epa.ms/KeyValueData10-9>)

```
using System.ComponentModel.DataAnnotations;           ❸
using KPCLib;
using PassXYZLib;
namespace PassXYZ.BlazorUI;

public class KeyValueData<T> : IKeyValue {
    [Required(ErrorMessage = "{0} cannot be empty.")]
    [Display(Name = "This field")]
    public string Key {                                ❶
        get {
            if (Data is Item item) { return item.Name; }
            if (Data is Field field) { return field.Key; }
            return string.Empty;
        }
    }
}
```

```

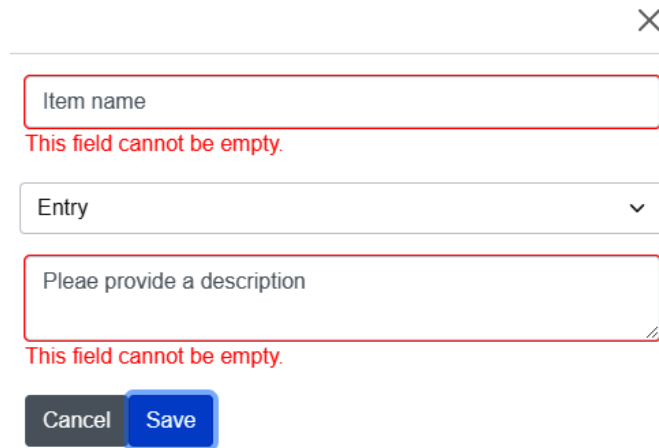
    }
    set {
        if (Data is Item item) { item.Name = value;
            IsChanged = true; }
        if (Data is Field field) { field.Key = value;
            IsChanged = true; }
    }
}
[Required(ErrorMessage = "{0} cannot be empty.")]
[Display(Name = "This field")]
public string Value { ❷
    get {
        if (Data is Item item) { return item.Notes; }
        if (Data is Field field) { return field.EditValue; }
        return string.Empty;
    }
    set {
        if (Data is Item item) { item.Notes = value;
            IsChanged = true; }
        if (Data is Field field) { field.EditValue = value;
            IsChanged = true; }
    }
}
public bool IsChanged { get; set; } = false;
public bool IsValid {
    get {
        return !(string.IsNullOrEmpty(Key) ||
            string.IsNullOrEmpty(Value));
    }
}
public T? Data { get; set; } ❸
public KeyValueData() {
    if (Data is Item item) { item = new NewItem(); }
    if (Data is Field field) { field = new NewField(); }
}
}

```

The `KeyValueData<T>` class is used to create the model instance for `EditForm`. It is a generic type and implements an `IKeyValue` interface, which defines properties that need to be implemented. In the `KeyValueData<T>` class, we define the `Key` ❶ and `Value` ❷ properties, `IsChanged`, `IsValid`, and `Data` ❸. We can see that we mark the `Key` and `Value` properties with data annotation `[Required]` and `[Display]` attributes. The `[Required]` attribute indicates the user must enter a value. The `[Display]` attribute defines the name to display for the property in the error message of data validation, as shown in *Figure 10.3*.

Both `[Required]` and `[Display]` are defined in the `System.ComponentModel.DataAnnotations` namespace ❹.

The `KeyValueData<T>` class is a generic class and takes a `T` type parameter to specify the actual data type for editing. The `Data` property ❸ is defined as the `T` type. We can pass either the `Item` or `Field` class as a type parameter to the `Data` property. The `Key` or `Value` property is bound to the property of the `Item` or `Field` instance stored in the `Data` property.



The screenshot shows a web form titled 'EditFormDialog' with a close button (X) in the top right corner. The form contains three input fields: a text box labeled 'Item name', a dropdown menu labeled 'Entry', and a text area labeled 'Please provide a description'. The 'Item name' and 'Please provide a description' fields are outlined in red, indicating they are required and currently empty. Below each of these fields is a red error message: 'This field cannot be empty.' At the bottom of the form are two buttons: 'Cancel' and 'Save'.

Figure 10.3: `EditFormDialog` (adding a new item)

We can see in *Figure 10.3* that the error messages will be displayed when users submit the form with an empty value in either the key or value field. To enable data validation, we also need to add the `<DataAnnotationsValidator />` component to `EditForm`, as shown here:

```
<EditForm class="row gx-2 gy-3" Model="@ModelData"
  OnValidSubmit="@HandleValidSubmit">
  <DataAnnotationsValidator />
  ...
</EditForm>
```

Using `EditForm`, it is quite easy to support data validation. To enable data validation using `EditForm`, follow these steps:

1. Bind `EditForm` to a model that uses data annotations attributes.
2. Add the `<DataAnnotationsValidator/>` component to `EditForm`.
3. Use built-in input components for the editing

There is a long list of built-in attribute-defined validations. You can find out more information about them in the Microsoft documentation in the *Further reading* section. Some of them are listed here:

- `[ValidateNever]`: Indicates that a property or parameter should be excluded from validation
- `[Compare]`: Validates that two properties in a model match
- `[EmailAddress]`: Validates that the property has an email format
- `[Phone]`: Validates that the property has a telephone number format
- `[Range]`: Validates that the property value falls within a specified range
- `[RegularExpression]`: Validates that the property value matches a specified regular expression
- `[Required]`: Validates that the field isn't null
- `[StringLength]`: Validates that a string property value doesn't exceed a specified length limit
- `[Url]`: Validates that the property has a URL format

With the introduction of built-in components, we can create a new Razor component, `EditFormDialog`, in the `PassXYZ.BlazorUI` project. The implementation of the `EditFormDialog<TItem>` component can be found in *Listing 10.10* and *Listing 10.11*:

Listing 10.10: `EditFormDialog.razor` (<https://epa.ms/EditFormDialog10-10>)

```
@namespace PassXYZ.BlazorUI
@typeparam TItem

<div class="modal fade" id=@Id tabindex="-1"
    aria-labelledby="ModelLabel" aria-hidden="true">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="ModelLabel">
                    @ModelData.Key</h5>
```

```

        <button type="button" class="btn btn-close"
            data-bs-dismiss="modal"
            aria-label="Close"></button>
    </div>
    <div class="modal-body">
        <EditForm class="row gx-2 gy-3" Model="@ModelData" ❶
            OnValidSubmit="@HandleValidSubmit">
            <DataAnnotationsValidator /> ❷
            @if (IsKeyEditingEnable) {
                <div class="form-group">
                    <InputText id="Name" class="form-control" ❸
                        @oninput="KeyHandler"
                        placeholder=@KeyPlaceHolder
                        @bind-Value="ModelData.Key" />
                    <ValidationMessage For="()=>ModelData.Key"/> ❺
                </div>
            }
            @ChildContent
            <div class="form-group">
                <InputTextArea id="Value" class="form-control" ❹
                    @oninput="KeyHandler"
                    placeholder=@ValuePlaceHolder
                    @bind-Value="ModelData.Value" />
                <ValidationMessage For="()=>ModelData.Value"/> ❺
            </div>
            <div class="col-12">
                <button type="button" class="btn btn-secondary"
                    data-bs-dismiss="modal"
                    @onclick="OnClickClose">
                    @CloseButtonText
                </button>
                <button type="submit" class="btn btn-primary"
                    data-bs-dismiss="@dataDismiss"
                    @onclick="OnClickSave">
                    @SaveButtonText
                </button>
            </div>
        </EditForm>
    </div>

```

```

        </div>
    </EditForm>
</div>
</div>
</div>
</div>

```

As we can see in the Razor markup in *Listing 10.10*, we build the `EditFormDialog<TItem>` component using Bootstrap's Modal. It includes the `EditForm` ❶ component in the modal body. The model bound to `EditForm` is `ModelData` of the `KeyValueData<TItem>` type defined in *Listing 10.11*. The `DataAnnotationsValidator` component ❷ is defined for the data validation.

We use `InputText` ❸ for key editing and `InputTextArea` ❹ for value editing. The `ValidationMessage` validation component ❺ is used to define the error message for validation. In `ValidationMessage`, we need to specify the field using the `For` property as follows:

```
<ValidationMessage For="() => ModelData.Key" />
```

The format of the error message is defined using the `ErrorMessage` property of the `[Required]` attribute as follows:

```

[Required(ErrorMessage = "{0} cannot be empty.")]
[Display(Name = "This field")]
public string Value { ... }

```

As we can see in *Listing 10.11*, component parameters and callback functions for the event handling are defined in `EditFormDialog.razor.cs`:

Listing 10.11: `EditFormDialog.razor.cs` (<https://epa.ms/EditFormDialog10-11>)

```

namespace PassXYZ.BlazorUI;

public partial class EditFormDialog<TItem> {
    [Parameter]
    public string Id { get; set; } = default!;
    [Parameter]
    public RenderFragment ChildContent { get; set; }
    [Parameter]
    public Action? OnClose { get; set; }
    [Parameter]

```

```
public Func<string, string, Task<bool>>? OnSaveAsync {
    get; set; }
[Parameter]
[NotNull]
public string CloseButtonText { get; set; } = "Cancel";
[Parameter]
[NotNull]
public string SaveButtonText { get; set; } = "Save";
[Parameter]
public KeyValueData<TItem> ModelData { get; set; }
[Parameter]
public string? KeyPlaceholder { get; set; }
[Parameter]
public string? ValuePlaceholder { get; set; }
bool _isKeyEditingEnable = false;
[Parameter]
public bool IsKeyEditingEnable ...
[Parameter]
public EventCallback<bool>
    IsKeyEditingEnableChanged{get; set;}
private string dataDismiss = string.Empty;

public EditFormDialog() {
    ModelData = new();
}
private void OnClickClose() {
    OnClose?.Invoke();
}
private void OnClickSave() {
    SetSaveButtonText();
}
private async Task HandleValidSubmit() {
    if (OnSaveAsync != null && ModelData.IsChanged) {
        await OnSaveAsync(ModelData.Key, ModelData.Value);
        ModelData.IsChanged = false;
    }
}
```

```

    }
    private void KeyHandler() {
        SetSaveButtonText(true);
    }
    private void SetSaveButtonText(bool changed = false) {
        if (ModelData == null) return;
        if (!ModelData.IsValid || changed) {
            dataDismiss = string.Empty;
            SaveButtonText = "Save";
        }
        else {
            dataDismiss = "modal";
            SaveButtonText = "Close";
        }
    }
}

```

We can refer to the component parameters of `EditFormDialog<TItem>` in *Table 10.2*:

Parameter	Description
Id	The ID of the dialog. This is used by Bootstrap.
ChildContent	Child content render fragments.
OnClose	Callback for the Close button.
OnSaveAsync	Callback for the Save button.
CloseButtonText	Text of the Close button.
SaveButtonText	Text of the Save button.
ModelData	Model data bound to EditForm.
KeyPlaceholder	A Key placeholder.
ValuePlaceholder	A Value placeholder.
IsKeyEditingEnable	Enable or disable key editing.
IsKeyEditingEnableChanged	Callback for two bindings of IsKeyEditingEnable.

Table 10.2: `EditFormDialog<TItem>` component parameters

After we have created the `EditFormDialog<TItem>` component, we have done all the optimization of our code. Let us review the final code of the `Items` (Listing 10.12) and `ItemDetail` (Listing 10.13) pages:

Listing 10.12: Items.razor (<https://epa.ms/Items10-12>)

```
@page "/"group"
@page "/"group/{SelectedItemId}"
@using System.Diagnostics
@using PassXYZLib

<Navbar ParentLink="@selectedItem?.GetParentLink()"
    Title="@Title"
    DialogId="@_dialogEditId"
    OnAddClick="@(() =>
        {_isNewItem=true;
        currentItem.Data=listGroupItem=_newItem;
        _newItem.Name="";_newItem.Notes="";})" />
<ListView Items="items">
    <Row Context="item">
        
        <a href="@item.GetActionLink()"
            class="list-group-item ...">
            <div class="d-flex"><div>
                <h6 class="mb-0">@item.Name</h6>
                <p class="mb-0 opacity-75">@item.Description</p>
            </div></div>
        </a>
        <Dropdown OnClick="@(() =>
            currentItem.Data=listGroupItem=item)">
            <MenuItem Id="@_dialogEditId" OnClick="@(() =>
                _isNewItem=false)">Edit</MenuItem>
            <MenuItem Id="@_dialogDeleteId">Delete</MenuItem>
        </Dropdown>
    </Row>
```

```

</ListView>

<EditFormDialog Id="@_dialogEditId"                                ❷
    ModelData="@currentItem" IsKeyEditingEnable="true"
    OnSaveAsync="UpdateItemAsync" KeyPlaceholder="Item name"
    ValuePlaceholder="Pleae provide a description">
    @if (_isNewItem) {
<InputSelect @bind-Value="_newItem.SubType"
    class="form-select">
        <option selected value=@ItemSubType.Group>
            @ItemSubType.Group</option>
        <option value=@ItemSubType.Entry>
            @ItemSubType.Entry</option>
        <option value=@ItemSubType.PxEntry>
            @ItemSubType.PxEntry</option>
        <option value=@ItemSubType.Notes>
            @ItemSubType.Notes</option>
        </InputSelect>
    }
</EditFormDialog>
<CascadingValue Value="@_dialogDeleteId" Name="Id">
<ConfirmDialog Title=@listGroupItem.Name                            ❸
    OnConfirmClick="DeleteItemAsync" />
</CascadingValue>

```

In the final code of `Items.razor`, we use `Navbar` ❶ to create the navigation bar. In the body of the page, we use a `ListView` ❷ component to display the list of items. For each item, the UI template is defined in the `Row` property. In the UI template, we display the item name, description, and context menu. In the context menu, we use the `Dropdown` ❸ component to support editing or deleting the item. For Add, Edit, or Delete actions, `EditorFormDialog` ❹ or `ConfirmDialog` ❺ is used to perform the respective actions.

In `ItemDetail.razor`, the code is similar to `Items.razor`:

Listing 10.13: `ItemDetail.razor` (<https://epa.ms/ItemDetail10-13>)

```

@page "/entry/{SelectedItemId}"
@namespace PassXYZ.Vault.Pages

```

```

<!-- Back button and title -->
<Navbar ParentLink="@selectedItem?.GetParentLink()" ①
    Title="@selectedItem?.Name" DialogId="@_dialogEditId"
    OnAddClick="@(() => {_isNewField=true;
        currentField.Data=listGroupField=_newField;_
        newField.Key="";_newField.Value="";})" />
<!-- List view with context menu -->
<ListView Items="fields"> ②
    <Row Context="field"> ③
        @if (field.ShowContextAction == null) {
            <span class="oi oi-pencil" aria-hidden="true"></span>
            <div class="d-flex gap-2 w-100
                justify-content-between">
                <div>
                    <h6 class="mb-0">@field.Key</h6>
                    <p class="mb-0">@field.Value</p>
                </div>
            </div>
            <Dropdown OnClick="@(() =>
                {currentField.Data=listGroupField=field;})">
                <MenuItem Id="@_dialogEditId" OnClick="@(() =>
                    _isNewField=false)">Edit</MenuItem>
                <MenuItem Id="@_dialogDeleteId">Delete</MenuItem>
                @if (field.IsProtected) {
                    <MenuItem OnClick="OnToggleShowPassword">
                        @(field.IsHide ? "Show":"Hide")</MenuItem>
                }
            </Dropdown>
        }
    </Row>
    <Footer>@( (MarkupString) notes)</Footer> ④
</ListView>

<!-- Editing Modal -->
<EditFormDialog Id="@_dialogEditId" ⑤
    ModelData="@currentField"

```

```

IsKeyEditingEnable=@_isNewField
OnSaveAsync="UpdateFieldAsync"
KeyPlaceholder="Field name"
ValuePlaceholder="Field content">
@if (_isNewField) {
    <div class="form-check">
        <input class="form-check-input" type="checkbox"
            @bind="listGroupField.IsProtected"
            id="flexCheckDefault">
        <label class="form-check-label"
            for="flexCheckDefault">
            Password
        </label>
    </div>
}
</EditFormDialog>
<!-- Deleting Modal -->
<CascadingValue Value="@_dialogDeleteId" Name="Id">
    <ConfirmDialog Title=@listGroupField.Key
        OnConfirmClick="DeleteFieldAsync" />
</CascadingValue>

```

⑥

There is a navigation bar using `Navbar` ①. We use `ListView` ② to display the fields of an entry, which include a Row template ③ and a Footer template ④. In the Row template, the content of the field is displayed, and each row has a context menu. The context menu includes the menu of three actions (Edit, Delete, and Show). The Edit action is associated with the `EditFormDialog` dialog ⑤ and the Delete action is associated with the `ConfirmDialog` dialog ⑥.

We now have a working password manager app built using Blazor UI. With this, we conclude this chapter and *Part 2* of this book.

Summary

In this chapter, we continue our work of optimizing our UI implementation by creating more Razor components. We introduced advanced topics such as templated components and using generic types in the Razor component. We also introduced Blazor built-in components and used these components to support data validation in our app. We developed an enhanced version of the key value editor using `EditForm`. With an understanding of the advanced topics about Razor components, we can now create more powerful Razor components.

We have completed *Part 2* of this book. We will move to *Part 3* of this book to introduce unit tests and deployment of .NET MAUI applications. We will learn how to implement unit tests for the .NET MAUI app in the next chapter, and how to deploy our app to different app stores in *Chapter 12, Preparing for Deployment in App Stores*.

Please stay tuned!

Further reading

- Dropdown is a Bootstrap component
- <https://getbootstrap.com/docs/5.1/components/dropdowns/>
- Modal is the Bootstrap modal component
- <https://getbootstrap.com/docs/5.1/components/modal/>
- ASP.NET Core Blazor forms and input components
- <https://learn.microsoft.com/en-us/aspnet/core/blazor/forms-and-input-components?view=aspnetcore-6.0>

Part 3:

Testing and Deployment

In *Part 1* of this book, we introduced .NET MAUI app development. In *Part 2*, we introduced .NET MAUI Blazor Hybrid app development. We now have two different versions of the password management app from *Part 1* and *Part 2*. Both XAML and Blazor Hybrid apps use the MVVM pattern in their design. When we introduced the MVVM pattern, we mentioned that we can test the view model and model separately. In *Part 3* of this book, we will introduce unit testing in .NET MAUI. Lastly, we will introduce the deployment of the .NET MAUI app.

This section comprises the following chapters:

- *Chapter 11, Developing Unit Tests*
- *Chapter 12, Deploying and Publishing in App Stores*

Developing Unit Tests

Testing is an important way to ensure software quality in modern software development. There are different types of testing involved in the software development lifecycle, such as unit testing, integration testing, and system testing. Unit testing is used to test software modules or components in an isolated environment. It is usually done by developers. With a well-planned unit test strategy, programming issues can be found at the earliest stage in the software development lifecycle, so unit testing is the most efficient and economical approach to ensuring the quality of your software. In .NET MAUI app development, we can reuse existing unit test frameworks or libraries in the .NET ecosystem. By using a test framework or library, we can speed up the unit test development. A good test framework is usually designed to easily integrate with a **continuous integration (CI)** and **continuous deployment (CD)** environment. In this chapter, we will introduce how to set up unit testing and run unit test cases as part of the .NET MAUI app development lifecycle.

We will cover the following topics in this chapter:

- Unit testing in .NET
- Razor component testing using bUnit

Technical requirements

To test and debug the source code in this chapter, you need to have Visual Studio 2022 installed on your PC or Mac. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI*, for the details.

The source code for this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter11>

The source code can be downloaded using the following Git command:

```
git clone -b chapter11 https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development PassXYZ.Vault2
```

Unit testing in .NET

To develop unit test cases, we usually use a unit test framework to improve efficiency. There are several unit test frameworks available in a .NET environment as follows:

- **Microsoft Test Framework (MSTest)** is shipped together with Visual Studio. The initial version of MSTest (V1) was not an open source product. The first release was shipped with Visual Studio 2005. Please refer to the book *Microsoft Visual Studio 2005 Unleashed* by Lars Powers and Mike Snell to find more information about MSTest (V1). Later, Microsoft made the new-release MSTest (V2) open source and hosted it on GitHub. The first MSTest (V2) release was available around 2017.
- **NUnit** is an open source testing framework ported from **JUnit**. It was the first unit test framework for .NET. The earliest releases were hosted at SourceForge in 2004. Please refer to the version 2.0 release note in the *Further reading* section. The most recent releases have been moved to GitHub.
- **xUnit** is a more modern and extensible framework developed by Jim Newkirk and Brad Wilson. They were the creators of NUnit, and they made many improvements to this new test framework compared to NUnit. Please refer to Jim's blog *Why Did we Build xUnit 1.0?* to find out more information about the improvements. The first stable release of xUnit was available around 2015.

They are all quite popular and can be chosen based on the project requirements. In this chapter, we will use xUnit to develop unit test cases since it is a newer framework with many improvements compared to NUnit.

No matter which unit test framework you choose, the process of unit test development is quite similar. The content in this chapter can still help you if you use a different framework in your project. Unit test cases can only run against a cross-platform target framework rather than platform-specific target frameworks. In this book, we use .NET 6.0, so the target framework of unit testing is `net6.0` instead of `net6.0-android` or `net6.0-ios`.

To develop a unit test for .NET MAUI, we will introduce the test case development for both XAML-based and Blazor-based apps. In both cases, we will use the MVVM pattern in the design. The unit test cases at the model layer are the same for both, but the testing in the view and the view model is quite different. For a XAML-based app, it is quite complicated to develop unit test cases for the view and the view model. In order to test the view model, we have to resolve the dependencies of XAML components. For example, in the XAML version of our app, we need to call `Shell` navigation methods in the view model as shown in the following code:

```
await Shell.Current.GoToAsync(  
    $"{{nameof(ItemsPage)}}?{{nameof(ItemsViewModel.ItemId)}}={{item  
        .Id}}";
```

To resolve dependencies, in `Xamarin.Forms`, there is an open source project, `Xamarin.Forms.Mocks`, which can help mock `Xamarin.Forms` components. We also need something similar to develop unit test cases for the view model in .NET MAUI XAML apps, but I cannot find any equivalent

at the moment. There is also a native user interface test framework, `Xamarin.UITest`, which is for Android and iOS, but this framework cannot be used in .NET MAUI yet. Regardless, `Xamarin.UITest` is not a cross-platform solution so we won't discuss it in this book.

For a Blazor Hybrid app, we have a good test library, **bUnit**, which can be used to test Razor components. We can develop unit test cases for the view, view model, and model layers for Blazor apps.

In this chapter, we will develop unit tests for the model layer first, which is common for both XAML and Blazor. After that, we will introduce unit test development for Blazor apps using **bUnit**. **bUnit** is a testing library that can be used with all three test frameworks (`xUnit`, `NUnit`, and `MSTest`).

Setting up the unit test project

To get our hands dirty, let us create a unit test project. We can create a `xUnit` project using either Visual Studio or the .NET command line:

1. To start with Visual Studio, we can add a new project to our current solution as shown in *Figure 11.1*:

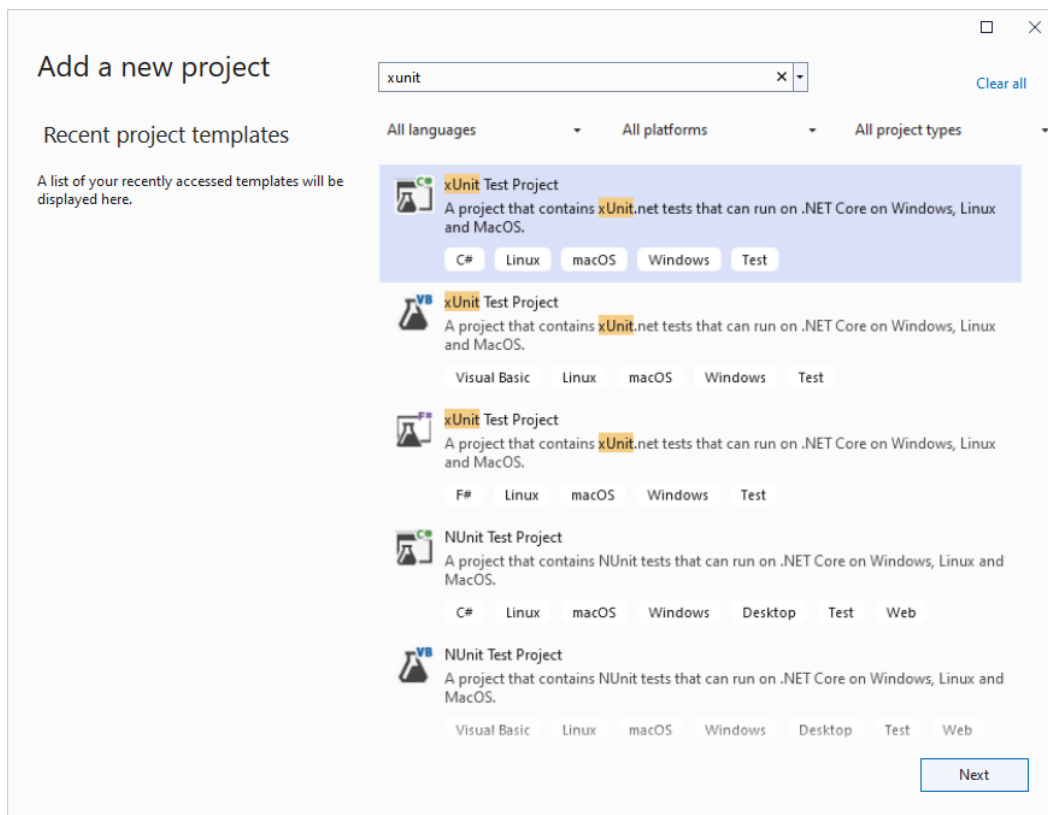


Figure 11.1 – Creating a `xUnit` project

2. We can type `xunit` into the search box and select **xUnit Test Project** for C#.
3. On the next screen, we can name the project `PassXYZ.Vault.Tests` and click on **Next**.
4. After that, please select the framework as `.NET 6.0` and click **Create**.

To create the project using a command line, we can create the folder first and use a `.NET` command to create the project as follows:

```
mkdir PassXYZ.Vault.Tests
cd PassXYZ.Vault.Tests
dotnet new xunit
dotnet test
```

Once we have created the project, we can use the `dotnet test` command to run the test cases. The default test case in the template will be executed. We will add test cases to this test project. The test targets are the components of the `PassXYZ.Vault` and `PassXYZ.BlazorUI` projects, so we need to add these two projects as reference projects. The target framework of `PassXYZ.BlazorUI` is `net6.0`, so we can add it directly. However, the target frameworks of `PassXYZ.Vault` are platform-specific, so we need to make some changes before we can add them as a reference in `PassXYZ.Vault.Tests`.

The project file of our password manager app is `PassXYZ.Vault.csproj`. We need to add `net6.0` as one of the target frameworks to this project file:

```
<TargetFrameworks>net6.0;net6.0-android;net6.0-ios;net6.0-maccatalyst</TargetFrameworks>
```

When we build project `PassXYZ.Vault` on the supported platforms, we expect an executable since it is an app. However, when we build `PassXYZ.Vault` for the `net6.0` target framework, we want to test it. `PassXYZ.Vault` should be generated as a library so that the test framework can use it to run test cases. In this case, we expect to build a file with a `.dll` extension instead of `.exe`, so we need to make the following change:

```
<OutputType Condition="'$(TargetFramework)' != 'net6.0'>
  Exe</OutputType>
```

In the preceding build setup, a condition is added to check the target framework for the output type. If the target framework is not `net6.0`, we will build output as an executable.

With these changes, we can add reference projects to `PassXYZ.Vault.Tests` by right-clicking on the solution node and selecting **Add -> Project Reference ...** or editing the project file for `PassXYZ.Vault.Tests` to add these lines:

```
<ItemGroup>
  <ProjectReference Include="..\PassXYZ.BlazorUI\PassXYZ.
    BlazorUI.csproj" />
  <ProjectReference Include="..\PassXYZ.Vault\PassXYZ.Vault.
    csproj" />
</ItemGroup>
```

To test the MAUI project, we also need to add the following configuration to the `PassXYZ.Vault.Tests` project:

```
<UseMaui>true</UseMaui>
```

Now, we have set up the xUnit project. Let us add our test cases.

Creating test cases to test the `IDataStore` interface

We will start to add test cases at the model layer first since the test case setup at the model layer is the same for both the XAML and Blazor versions of our app.

At the model layer, the major implementation is in the `PassXYZLib` library – you may refer to the source code of `PassXYZLib` to find more about the unit test cases at the model layer:

<https://github.com/shugaoye/PassXYZLib>

In our app, `IDataStore` is the interface to export `PassXYZLib`, so let's add test cases to our test interface, `IDataStore`. To test the `IDataStore` interface, we can create a new test class, `DataStoreTests`, in the `PassXYZ.Vault.Tests` project. We can start to add a new test case to test the case by adding an item as follows:

```
public class DataStoreTests
{
    [Fact]
    public async void Add_Item()
    {
        // Arrange
        IDataStore<Item> datastore = new MockDataStore();
        ItemSubType itemType = ItemSubType.Entry;
```

```

        // Act ❸
        var newItem = datastore.CreateNewItem(itemSubType);
        newItem.Name = $"{itemSubType.ToString()}01";
        await datastore.AddItemAsync(newItem);
        var item = datastore.GetItem(newItem.Id);

        // Assert ❹
        Assert.Equal(newItem.Id, item.Id);
    }
}

```

xUnit uses attributes to inform the framework about test case setup. In this test case, we use the [Fact] attribute, **❶**, to mark this method as a test case. To define a test case, we can use a common pattern – **Arrange**, **Act**, and **Assert**:

- **Arrange** **❷** – We will prepare all necessary setup for the test. To add an item, we need to initialize the IDataStore interface first, and then we will define a variable to hold the item type.
- **Act** **❸** – We execute the methods that we want to test, which are CreateNewItem() and AddItemAsync().
- **Assert** **❹** – We check the result that we expected. In our case, we try to retrieve the new item using item.Id. After that, we check to ensure that the item ID retrieved is the same as what we expected.

As you may have noticed, we tested the Entry type in the previous test case. The Entry type is only one of the item types – we have many. To test all of them, we need to create many test cases. xUnit supports another test case type, [Theory], which helps us to test different scenarios with one test case.

We can use the “delete an item” test case to demonstrate how to test different scenarios in one test case with the [Theory] attribute. In this test case, we can delete an item in different item types in one test case:

```

public class DataStoreTests
{
    ...
    [Theory] ❶
    [InlineData(ItemSubType.Entry)] ❷
    [InlineData(ItemSubType.Group)]
    [InlineData(ItemSubType.Notes)]
    [InlineData(ItemSubType.PxEntry)]
    public async void Delete_Item(ItemSubType itemSubType)

```

```
{
    // Arrange
    IDataStore<Item> datastore = new MockDataStore();
    var newItem = datastore.CreateNewItem(itemSubType); ③
    newItem.Name = $"{itemSubType.ToString()}01";
    await datastore.AddItemAsync(newItem);

    // Act
    bool result = await
        datastore.DeleteItemAsync(newItem.Id);          ④
    Debug.WriteLine($"Delete_Item: {newItem.Name}");

    // Assert
    Assert.True(result);                                ⑤
}
...
}
```

When we create a test case using the `[Theory]` attribute, ①, we can pass different item types using the `itemSubType` parameter. The value of the `itemSubType` argument is defined using the `[InlineData]` attribute, ②.

To arrange test data, we create a new item using the `itemSubType` argument, ③. Then, we execute the `DeleteItemAsync()` method, ④, which is the one that we want to test.

Finally, we check the return value, ⑤. If the item is deleted successfully, the result is true. Otherwise, the result is false.

We have learned how to create a test case using the `[Fact]` attribute and how to cover different scenarios using the `[Theory]` attribute. Let's discuss more topics in test case development in the next section.

Sharing context between tests

In our previous test cases, we created a new `IDataStore` instance for each test case. Can we share one `IDataStore` instance instead of creating the same instance every time? We can reduce duplication by sharing the test setup among a group of test cases in xUnit.

There are three ways to share the setup and cleanup code between tests in xUnit:

- **Constructor and Dispose** – we can use a class constructor to share the setup and cleanup code without sharing instances
- **Class Fixture** – we can use a fixture to share object instances in a single class
- **Collection Fixtures** – we can use collection fixtures to share object instances in multiple test classes

Sharing using a constructor

To remove the duplicated setup code from the previous tests, we can move the creation of the `IDataStore` instance to the constructor of the `DataStoreTests` test class as follows:

```
public class DataStoreTests
{
    IDataStore<Item> datastore;
    public DataStoreTests()
    {
        datastore = new MockDataStore();
        Debug.WriteLine("DataStoreTests: Created");
    }
    ...
}
```

In this code, we added a private member variable, `datastore`, and created an instance of `IDataStore` in the constructor of `DataStoreTests`. We also added a debug output so we can monitor the creation of the `IDataStore` interface. Let us debug the execution of the `DataStoreTests` class so we can see the debug output here:

```
DataStoreTests: Created
Delete_Item: Entry01
DataStoreTests: Created
Delete_Item: Group01
DataStoreTests: Created
Delete_Item: PxEntry01
DataStoreTests: Created
Delete_Item: Notes01
DataStoreTests: Created
Create_Item: PxEntry
```

```
DataStoreTests: Created
Create_Item: Group
DataStoreTests: Created
Create_Item: Entry
DataStoreTests: Created
Create_Item: Notes
DataStoreTests: Created
Add_Item: Done
```

We can see from the debug output that a `DataStoreTests` class is created for each test case. There is no difference whether we create the instance of `IDataStore` inside the test method or in the constructor. All the test cases are still isolated from each other. When we use the `[Theory]` attribute to test different scenarios with one method, each of them looks like a separate test case at runtime. To understand this better, we can use `dotnet` command to list all the tests defined:

```
dotnet test -t
Determining projects to restore...
All projects are up-to-date for restore.
Microsoft (R) Test Execution Command Line Tool Version 17.3.0
(x64)
Copyright (c) Microsoft Corporation. All rights reserved.
The following Tests are available:
    PassXYZ.Vault.Tests.DataStoreTests.Add_Item
    PassXYZ.Vault.Tests.DataStoreTests.Delete_Item(itemSubType:
        Entry)
    PassXYZ.Vault.Tests.DataStoreTests.Delete_Item(itemSubType:
        Group)
    PassXYZ.Vault.Tests.DataStoreTests.Delete_Item(itemSubType:
        Notes)
    PassXYZ.Vault.Tests.DataStoreTests.Delete_Item(itemSubType:
        PxEntry)
    PassXYZ.Vault.Tests.DataStoreTests.Create_Item(itemSubType:
        Entry)
    PassXYZ.Vault.Tests.DataStoreTests.Create_Item(itemSubType:
        Group)
    PassXYZ.Vault.Tests.DataStoreTests.Create_Item(itemSubType:
        Notes)
```

```
PassXYZ.Vault.Tests.DataStoreTests.Create_Item(itemSubType:
    PxEntry)
```

We can see that each parameter defined by the [InlineData] attribute is shown as a separate test case. They are all isolated test cases at runtime.

After we list all tests, we can selectively execute them using the dotnet command.

If we want to run all the tests in the DataStoreTests class, we can use this command:

```
dotnet test --filter DataStoreTests
```

If we want to run Add_Item tests only, we can use this command:

```
dotnet test --filter DataStoreTests.Add_Item
```

As we can see from the debug output, even though we created an instance of IDataStore in the constructor, the instance is re-created for each test. The instances created in the test class constructor won't be shared across tests. Even though the effect is still the same, the code looks more concise.

However, in some cases, we do want to share instances across tests. We can do so using class fixtures. Let's look at these cases in the next section.

Sharing using class fixtures

When we use a tool in all test cases, we may want to share the setup in all test cases instead of creating the same one every time. Let's use a logging function as an example to explain this.

To have a test report, we want to create a test log to monitor the execution of unit tests. There is a library called Serilog that can be used for this purpose. We can log messages to different channels using Serilog. To use Serilog, we need to set it up first and clean up it after all the tests have been executed. In this case, we want to share an instance of Serilog between all the tests instead of creating one for each test. With this setup, we can generate one log file for all the tests instead of multiple log files for each test.

To use Serilog, we need to add the Serilog package to the project first. To do that, we can run the following dotnet commands in the project's PassXYZ.Vault.Tests folder:

```
dotnet add package Serilog
dotnet add package Serilog.Sinks.File
```

After adding the Serilog libraries to the project, we can create a class fixture, SerilogFixture, for demonstration purposes:

```

public class SerilogFixture : IDisposable {
    public ILogger Logger { get; private set; }
    public SerilogFixture() {
        Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .WriteTo.File(@"logs\xunit_log.txt")
            .CreateLogger();
        Logger.Debug("SerilogFixture: initialized");
    }
    public void Dispose() {
        Logger.Debug("SerilogFixture: closed");
        Log.CloseAndFlush();
    }
}

public class DataStoreTests : IClassFixture<SerilogFixture>
{
    IDataStore<Item> datastore;
    SerilogFixture serilogFixture;

    public DataStoreTests(SerilogFixture fixture) {
        serilogFixture = fixture;
        datastore = new MockDataStore();
        serilogFixture.Logger.Debug("DataStoreTests: Created");
    }
    [Fact]
    public async void Add_Item() ...

    ...
}

```

If we want to use class fixtures, we can create them using the following steps:

1. We can create a new class as the fixture class and add the setup code to the constructor. Here, we created a fixture class, `SerilogFixture`, ❶, and initialized the `ILogger` interface, ❷, in the constructor.
2. Because we need to clean up the setup after the test case execution, we need to implement the `IDisposable` interface for the fixture class and put the cleanup code in the `Dispose()`

method. We implemented `IDisposable` in `SerilogFixture` and called the `Serilog` function, `Log.CloseAndFlush()`, ❸, in the `Dispose()` method.

3. To use the fixture, the test case needs to implement the `IClassFixture<T>` interface. We implemented this in the `DataStoreTests` test class, ❹.
4. To access the fixture instance, we can add it as a constructor argument and it will be provided automatically. In the constructor of `DataStoreTests`, ❺, we assign the argument to the private member variable, `serilogFixture`, ❻. In test cases, we can access `Serilog` using this variable.

To verify this setup, we replaced all our debug output with `Serilog Debug`. After executing the tests in `DataStoreTests`, we can see the log messages here in the `xunit_log.txt` log file:

```
2022-08-28 10:25:39.273 +08:00 [DBG] SerilogFixture:
initialized
2022-08-28 10:25:39.332 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.350 +08:00 [DBG] Delete_Item: Entry01
2022-08-28 10:25:39.355 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.355 +08:00 [DBG] Delete_Item: Group01
2022-08-28 10:25:39.356 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.357 +08:00 [DBG] Delete_Item: PxEntry01
2022-08-28 10:25:39.358 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.358 +08:00 [DBG] Delete_Item: Notes01
2022-08-28 10:25:39.359 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.359 +08:00 [DBG] Create_Item: PxEntry
2022-08-28 10:25:39.360 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.360 +08:00 [DBG] Create_Item: Group
2022-08-28 10:25:39.361 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.361 +08:00 [DBG] Create_Item: Entry
2022-08-28 10:25:39.362 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.362 +08:00 [DBG] Create_Item: Notes
2022-08-28 10:25:39.362 +08:00 [DBG] DataStoreTests: Created
2022-08-28 10:25:39.364 +08:00 [DBG] Add_Item: Done
2022-08-28 10:25:39.367 +08:00 [DBG] SerilogFixture: closed
```

As we expected, the `SerilogFixture` class is initialized just once, and the instance can be used in all the tests in `DataStoreTests`, compared to the `IDataStore` interface being initialized for each test.

Sharing using collection fixtures

Using class fixtures, as shown in the previous section, we can share the test setup context in one test class. There are also cases in which we might want to share the test setup in multiple test classes. We can do so using collection fixtures.

In our case of Serilog, we can use it in many test classes as well so that we can see all the log messages in one log file. To use one Serilog setup for all the test classes, we can implement collection fixtures in our project. To use collection fixtures, we can create two new classes, `SerilogFixture` and `SerilogCollection`, in the `PassXYZ.Vault.Tests` project as shown in *Listing 11.1*:

Listing 11.1: SerilogFixture.cs (<https://epa.ms/SerilogFixture11-1>)

```
namespace PassXYZ.Vault.Tests;

public class SerilogFixture : IDisposable {
    public ILogger Logger { get; private set; }

    public SerilogFixture() {
        Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .WriteTo.File(@"logs\xunit_log.txt")
            .CreateLogger();

        Logger.Debug("SerilogFixture: initialized");
    }

    public void Dispose() {
        Logger.Debug("SerilogFixture: closed");
        Log.CloseAndFlush();
    }
}

[CollectionDefinition("Serilog collection")]
public class SerilogCollection : ICollectionFixture<SerilogFixture>
{
}
```

1**2**

We can follow the steps below to implement collection fixtures:

1. We create a new class file, `SerilogFixture.cs`, and put both `SerilogFixture` and `SerilogCollection` into this file.
2. We decorate the collection definition class, `SerilogCollection`, with the `[CollectionDefinition]` attribute, ❶. Then, we give it a unique name that can be used to identify the test collection.
3. The collection definition class, `SerilogCollection`, needs to implement the `ICollectionFixture<T>` interface, ❷.

To use a collection fixture, we can make the following changes to our test classes:

1. We can add the `[Collection]` attribute to all the test classes that will be part of the collection. We assign `Serilog collection` as a name to the test collection definition in the attribute. In our case, as we can see in *Listing 11.2*, we add a `[Collection("Serilog collection")]` attribute, ❶, to the `DataStoreTests` class.
2. To access the fixture instance, we can do the same as in the previous section with class fixtures and add it as a constructor argument. Then, it will be provided automatically. In the constructor of `DataStoreTests`, we assign a fixture argument to the `serilogFixture` variable, ❷:

Listing 11.2: DataStoreTests.cs (<https://epa.ms/DataStoreTests11-2>)

```
namespace PassXYZ.Vault.Tests;

[Collection("Serilog collection")] ❶
public class DataStoreTests {
    IDataStore<Item> datastore;
    SerilogFixture serilogFixture;

    public DataStoreTests(SerilogFixture fixture) {
        datastore = new MockDataStore();
        serilogFixture = fixture; ❷
        serilogFixture.Logger.Debug("DataStoreTests
            initialized");
    }
    [Fact]
    public async void Add_Item() {
        // Arrange
        ItemSubType itemSubType = ItemSubType.Entry;
```

```

// Act
var newItem = datastore.CreateNewItem(itemSubType);
newItem.Name = $"{itemSubType.ToString()}01";
await datastore.AddItemAsync(newItem);
var item = datastore.GetItem(newItem.Id);

// Assert
Assert.Equal(newItem.Id, item.Id);
serilogFixture.Logger.Debug("Add_Item done");
}

[Theory]
[InlineData(ItemSubType.Entry)]
[InlineData(ItemSubType.Group)]
[InlineData(ItemSubType.Notes)]
[InlineData(ItemSubType.PxEntry)]
public async void Delete_Item(ItemSubType itemSubType)...

[Theory]
[InlineData(ItemSubType.Entry)]
[InlineData(ItemSubType.Group)]
[InlineData(ItemSubType.Notes)]
[InlineData(ItemSubType.PxEntry)]
public void Create_Item(ItemSubType itemSubType) ...
}

```

With these examples, we have introduced how to create unit tests in the model layer. The knowledge that we have gained so far can be used in unit testing for other .NET applications as well.

So far, we have concluded the introduction of the model layer unit test. In the next part of this chapter, we will explore the Razor component unit test using the bUnit library.

Razor component testing using bUnit

In .NET MAUI development, we don't really have a good unit test framework for XAML-based UI components, but we do have one for Blazor. bUnit is an excellent test library that can be used for the unit test development of Razor components. With the bUnit library, we can develop unit test cases for Razor components using xUnit, NUnit, or MSTest. We will use xUnit with bUnit for the rest of the chapter. The structure of unit test cases using bUnit is similar to the xUnit test cases that we introduced in the previous section.

The test targets in the rest of this chapter are the following Razor components that we created in the second part of this book:

- Razor components in the `PassXYZ.BlazorUI` project
- Razor components in the `PassXYZ.Vault` project

To test Razor components using bUnit, we need to change the project configuration of `PassXYZ.Vault.Tests`.

Changing project configuration for bUnit

To set up the test environment, we need to add the bUnit and Moq packages and, update the SDK type. We can make the following changes to the xUnit `PassXYZ.Vault.Tests` testing project:

1. Add bUnit to the project.

To add the bUnit library to the project, we can change to the project folder first and execute the following command from a console:

```
cd PassXYZ.Vault.Tests
dotnet add package bunit
```

2. We also need to add the Moq package, which is a mocking library that we will use in the test setup:

```
dotnet add package Moq
```

3. Change the project configuration.

To test the Razor components, we also need to change the project's SDK to `Microsoft.NET.Sdk.Razor`.

In the `PassXYZ.Vault.Tests.csproj` project file, we need to replace the following line:

```
<Project Sdk="Microsoft.NET.Sdk">
```

We will do so with this:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

Once we have the project configuration ready, we can create a simple unit test case using bUnit to test our Razor components.

Creating a bUnit test case

In our `PassXYZ.Vault` app, we have two kinds of Razor components that can be tested. The shared Razor components reside in the `PassXYZ.BlazorUI` project. They are generic Razor components that can be used in different projects. The other set of Razor components is the one in the `Pages` folder of the `PassXYZ.Vault` project. They are specific to the `PassXYZ.Vault` app and they use shared components from the `PassXYZ.BlazorUI` project.

To test the Razor components in the `PassXYZ.BlazorUI` project, we can test each component separately. These test cases are unit test cases. The Razor components in the `Pages` folder of the `PassXYZ.Vault` project are UI pages. These pages use UI components from other packages, so they have more dependencies. These test cases can be considered integration test cases.

We can create a test case for the `ModalDialog` Razor component in the `PassXYZ.BlazorUI` project first. To test `ModalDialog`, we can create a xUnit test class, `ModalDialogTests`, as shown in *Listing 11.3*:

Listing 11.3: `ModalDialogTests.cs` (<https://epa.ms/ModalDialogTests11-3>)

```
namespace PassXYZ.Vault.Tests {
    [Collection("Serilog collection")]
    public class ModalDialogTests : TestContext {
        SerilogFixture serilogFixture;
        public ModalDialogTests(SerilogFixture serilogFixture) {
            this.serilogFixture = serilogFixture;
        }
        [Fact]
        public void ModalDialogInitTest() {
            string title = "ModalDialog Test";
            var cut = RenderComponent<ModalDialog>(
                parameters => parameters.Add(p => p.Title, title)
                .Add(p => p.CloseButtonText, "Close")
                .Add(p => p.SaveButtonText, "Save"));
            cut.Find("h5").TextContent.MarkupMatches(title);
            serilogFixture.Logger.Debug("ModalDialogInitTest:
                done");
        }
        ...
    }
}
```

As we can see in the `ModalDialogTests` unit test class, it is very similar to the unit test class that we created for the model layer. We reuse the collection fixture that we created before and initialized it in the constructor. In the `ModalDialogInitTest` test case, we still use the **Arrange**, **Act**, and **Assert** pattern to implement the test case.

All `bUnit` test classes inherit from `TestContext` ❶. In the **Arrange** phase, we initialize a local `title` variable, ❷, with a defined string. In the **Act** phase, we call a generic method, `RenderComponent<T>`, ❸, and use the `ModalDialog` type as the type parameter. We pass the `title` variable, ❹, as the component parameter. The result of `RenderComponent<T>` is stored in the `cut` variable. In the **Assert** phase, we verify that the title text after rendering is the same as the argument that we pass to it using the `Find()` `bUnit` method, ❺. The `Find()` `bUnit` method can be used to find any HTML tag. In `ModalDialog`, the title is rendered as a `<h5>` HTML tag.

In the `ModalDialogInitTest` test case, we can see the structure of the `bUnit` tests. In `bUnit` tests, we render the component under test first. The result of the rendering is stored in the `cut` variable, ❸. It is an instance of the `IRenderedComponent` interface. We can verify the result by referring to the properties or calling the methods of the `IRenderedComponent` instance.

When Razor components are rendered in `TestContext`, they have the same lifecycle as any other Razor component. We can pass parameters to components under test, and they can produce output, similar to what happens in a browser.

When we render the `ModalDialog` component in the preceding example, we can pass component parameters to it using the `Add()` method of a parameter builder of the `ComponentParameterCollectionBuilder<TComponent>` type.

We may not have a problem rendering simple components using C# code. However, we usually need to pass multiple parameters to a component, and it is not convenient to do so in C# code. With `bUnit`, we can develop test cases in Razor files, which can bring a much better experience in unit test development.

Creating test cases in Razor files

To create tests in Razor markup files directly, we can declare components using Razor markup as we use them in a Razor page. In this way, we don't have to call Razor components in the C# code and pass parameters using function calls. For a Razor page, we can render Razor components using **Razor templates**.

We can demonstrate how to create tests in Razor markup files by creating test cases for a more complicated `EditorDialog` component. We created the `EditorDialog` component in *Chapter 9, Razor Components and Data Binding*. In *Listing 11.4*, let's review the unit tests for it:

Listing 11.4: EditorDialogTests.razor (<https://epa.ms/EditorDialog-Tests11-4>)

```

@inherits TestContext ❶

<h3>EditorDialogTests</h3>

@code {
    bool _isOnCloseClicked = false;
    string _key = string.Empty;
    string _value = string.Empty;
    string updated_key = "key updated";
    string updated_value = "value updated";
    void OnSaveClicked(string key, string value) {
        _key = key; _value = value;
    }
    void OnCloseHandler() {
        _isOnCloseClicked = true;
    }

    [Fact]
    public void EditorDialog_Init_WithoutArgument() ...
    [Fact]
    public void Edit_OnClose_Clicked() {
var cut = Render(@<EditorDialog Key="@_key"
    Value="@_value"
    OnSave=@OnSaveClicked OnClose=@OnCloseHandler>
        </EditorDialog>); ❷
        cut.Find("button[class='btn btn-secondary']").Click(); ❸
        Assert.True(_isOnCloseClicked); ❹
    }

    [Fact]
    public void Edit_With_KeyEditingEnabled() { ❺
var cut = Render(@<EditorDialog Key="@_key"
    Value="@_value"
    IsKeyEditingEnable="true" OnSave=@OnSaveClicked>
        </EditorDialog>);
        cut.Find("input").Change(updated_key);
        cut.Find("textarea").Change(updated_value);
    }
}

```

```

        cut.Find("button[type=submit]").Click();
        Assert.Equal(_key, updated_key);
        Assert.Equal(_value, updated_value);
    }
    [Fact]
    public void Edit_With_KeyEditingDisabled() ...
}

```

We can create a new Razor component, `EditorDialogTests`, in the `PassXYZ.Vault.Tests` project. Since it is a bUnit test class, it is a child class of `TestContext`, ❶. In this class, we create test cases in a code block using Razor templates.

We can review the `Edit_OnClose_Clicked` test case first. In this test case, we render the `EditorDialog` component first and after that, we test the close button.

To render the `EditorDialog` component, we call the `Render()` method, ❷, of `TestContext`. Compared to the previous example, here, we can render the Razor markup directly instead of calling the C# function. The Razor markup that we use here is called **Razor templates** and you can find more information about it in this Microsoft document:

<https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#razor-templates-1>

Razor templates can be defined in the following format:

```
@<{HTML tag}>...</{HTML tag}>
```

It consists of an `@` symbol and a pair of open and closed HTML tags. Razor templates can be used in the code block of the Razor file. It cannot be used in a C# or C# code-behind file.

Using this format, we can specify a snippet of the Razor markup as the parameter of a C# function. The snippet of the Razor markup is a Razor template and its data type is `RenderFragment` or `RenderFragment<TValue>`. In *Listing 11.4*, we pass parameters to `EditorDialog` using Razor templates, as we can see in the following code:

```

var cut = Render(@<EditorDialog Key="@_key" Value=
    "@_value"
    OnSave=@OnSaveClicked OnClose=@OnCloseHandler>
    </EditorDialog>);

```

After `EditorDialog` is rendered, we can find the close button and simulate the click action, ❸:

```
cut.Find("button[class='btn btn-secondary']").Click();
```

In the `OnCloseHandler` event handler, the `_isOnCloseClicked` variable, ❹, is set to `true` so that we can assert the result.

In the `Edit_With_KeyEditingEnabled` test case, ❺, after the component is rendered, we can simulate user interactions to set the key and value fields in the component. After that, we can simulate clicking on the save button as we can see here:

```
cut.Find("input").Change(updated_key);  
cut.Find("textarea").Change(updated_value);  
cut.Find("button[type=submit]").Click();
```

When the button is clicked, the event handler is invoked. In the `OnSaveClicked` event handler, we save the changed key and value so we can assert the result:

```
Assert.Equal(_key, updated_key);  
Assert.Equal(_value, updated_value);
```

As we can see from these two test cases, we can design bUnit tests much more easily by creating tests in a Razor file. We can render components using Razor templates and we can also trigger various user interactions to test the components interactively.

Razor templates are a great tool to help us combine Razor markup and C# code so that we can leverage the best features from both worlds. However, we have a limitation when we use Razor templates. Let's see how to overcome it in the next section.

Using the `RenderFragment` delegate

Even though Razor templates can help simplify the test setup, there is a limitation, particularly in a complicated test case setup. In a complicated test case, the Razor templates can be very long. If we want to reuse the same Razor templates in another test case, we need to copy them to the new test case. We may have to create a lot of duplicated code, and this is the limitation of Razor templates.

In this case, we can use a `RenderFragment` delegate. As its name indicates, it is the delegate type of `RenderFragment` or `RenderFragment<TValue>`. The data type of Razor templates is `RenderFragment` or `RenderFragment<TValue>`. A `RenderFragment` delegate is the delegate type for Razor templates.

You can find more information about the `RenderFragment` delegate in the following Microsoft document:

<https://learn.microsoft.com/en-us/aspnet/core/blazor/performance?view=aspnetcore-3.1#define-reusable-renderfragments-in-code-2>

To demonstrate how to use the `RenderFragment` delegate, let's set up a more complex test for the `EditorDialog` component. `EditorDialog` can be used to edit either `Item` or `Field`. We can use an item-editing case to show how to use the `RenderFragment` delegate.

We can create a new test class, `ItemEditTests`, in the `PassXYZ.Vault.Tests` project. To separate the Razor markup and C# code, we can split the `ItemEditTests` test class into a Razor file (`ItemEditTests.razor`) and a C# code-behind file (`ItemEditTests.razor.cs`). We can declare the markup for testing in the Razor file, as shown in *Listing 11.5*:

Listing 11.5: `ItemEditTests.razor` (<https://epa.ms/ItemEditTests11-5>)

```
@inherits TestContext
@namespace PassXYZ.Vault.Tests

<h3>ItemEditTests</h3>
@code {
    private RenderFragment _editorDialog => __builder =>
    {
        <CascadingValue Value="@_dialogId" Name="Id">
        <EditorDialog IsKeyEditingEnable=@isNewItem
            OnSave=@OnSaveClicked Key=@testItem.Name
            Value=@testItem.Notes>
            @if (isNewItem) {
                <select id="itemType" @bind="testItem.ItemType"
                    class="form-select" aria-label="Group">
                    <option selected value="Group">Group</option>
                    <option value="Entry">Entry</option>
                    <option value="PxEntry">PxEntry</option>
                    <option value="Notes">Notes</option>
                </select>
            }
        </EditorDialog>
        </CascadingValue>
    }
}
```

```
};
}
```

We define a `RenderFragment` delegate, `_editorDialog`, in the `@code` block of `ItemEditTests.razor`. The `RenderFragment` delegate must accept a parameter called `__builder` of the `RenderTreeBuilder` type. In the markup code, we can access the variables defined in the test class.

Now let's look at the usage of `_editorDialog` in the C# code-behind file in *Listing 11.6*:

Listing 11.6: ItemEditTests.razor.cs (<https://epa.ms/ItemEditTests11-6>)

```
namespace PassXYZ.Vault.Tests;

[Collection("Serilog collection")]
public partial class ItemEditTests : TestContext {
    readonly SerilogFixture serilogFixture;
    bool isNewItem { get; set; } = false;
   NewItem testItem { get; set; }
    string _dialogId = "editItem";
    string updated_key = "Updated item";
    string updated_value = "This item is updated.";
    public ItemEditTests(SerilogFixture fixture) {
        testItem = new() {
            Name = "New item",
            Notes = "This is a new item."
        };
        serilogFixture = fixture;
    }
    void OnSaveClicked(string key, string value) {
        testItem.Name = key; testItem.Notes = value;
    }
    [Fact]
    public void Edit_New_Item() {
        isNewItem = true;
        var cut = Render(_editorDialog);
        cut.Find("#itemType").Change("Entry");
        cut.Find("input").Change(updated_key);
        cut.Find("textarea").Change(updated_value);
```

❶

```

        cut.Find("button[type=submit]").Click();
        Assert.Equal(updated_key, testItem.Name);
        Assert.Equal(updated_value, testItem.Notes);
    }
    [Fact]
    public void Edit_Existing_Item() {
        isNewItem = false;
        var cut = Render(_editorDialog);
        var ex = Assert.Throws<ElementNotFoundException>(() =>
            cut.Find("#itemType").Change("Entry"));
        Assert.Equal("No elements were found that matches the
            selector '#itemType'", ex.Message);
        cut.Find("textarea").Change(updated_value);
        cut.Find("button[type=submit]").Click();
        Assert.Equal(updated_value, testItem.Notes);
    }
}

```

Since `_editorDialog` defines the Item editing, we can implement multiple test cases against `_editorDialog`. We can see that we render `_editorDialog`, ❶, for multiple test cases, such as `Edit_New_Item` and `Edit_Existing_Item`. Using the `RenderFragment` delegate, our testing code looks much more elegant and cleaner. If we did not go this way, we would need to repeat long markup code in multiple places. Using C# code directly may have even led to more duplicated code.

In both test cases, we follow a similar process to testing `EditorDialog` by setting values and then clicking on the **Save** button. In the markup code, we have a `<select>` tag defined. We can change the option, ❷, of the `<select>` tag in the test code. This `<select>` tag is rendered conditionally referring to the value of the `isNewItem` variable. In the `Edit_Existing_Item` test, we can also test the negative case when the `isNewItem` variable, ❸, is set to `false`. In this case, an exception is thrown since the `<select>` tag is not rendered. We can see that `bUnit` can also be used to test negative cases by verifying the content of exception, ❹.

We created `bUnit` tests for the shared components in the `PassXYZ.BlazorUI` project in the previous examples. Since these shared components are reusable building blocks for a high-level UI, most of them declare many component parameters. The `RenderFragment` delegate or Razor templates can help to simplify the test setup.

If we move to the Razor pages in the `Pages` folder of the `PassXYZ.Vault` project, `Items`, `ItemDetail`, or `Login` are Razor components as well, but they are not designed for reuse. They are Razor pages with route templates defined and they don't have many component parameters defined.

The component parameters declared in these Razor pages are used for routing purposes. When we design test cases for these Razor pages, we can implement tests in a C# class rather than Razor files.

Testing Razor pages

In the process of development testing for Razor pages, we will learn about some very useful bUnit features. We won't be able to review all the tests of Razor pages in our app, so we will use `ItemDetail` as an example. `ItemDetail` is a Razor page for displaying the content of a password entry. There is a route defined for it:

```
@page "/entry/{SelectedItemId}"
```

When we want to display the `ItemDetail` page, we need to pass the `Id` info for an `Item` instance to it, and this instance cannot be a group. The initialization of the `ItemDetail` page is done in the `OnParametersSet()` life cycle method as we can see here:

```
protected override void OnParametersSet() {
    base.OnParametersSet();
    if (SelectedItemId != null) {
        selectedItem = DataStore.GetItem(SelectedItemId, true);
        if (selectedItem == null) {
            throw new InvalidOperationException(                ❷
                "ItemDetail: entry cannot be found with SelectedItemId");
        }
        else {
            if (selectedItem.IsGroup) {
                throw new InvalidOperationException(                ❸
                    "ItemDetail: SelectedItemId should not be group here.");
            }
            else {
                fields.Clear();                                ❹
                List<Field> tmpFields = selectedItem.GetFields();
                foreach (Field field in tmpFields) {
                    fields.Add(field);
                }
                notes = selectedItem.GetNotesInHtml();
            }
        }
    }
}
```

```

    else {
        throw new InvalidOperationException(
            "ItemDetail: SelectedItemId is null");
    }
}

```

We will develop an `ItemDetailTests` test class to cover all the execution paths in `OnParametersSet()`. To cover all the execution paths, we can find the following test cases:

- **Test case 1:** Initialize the `ItemDetail` instance without a selected item Id. We will get an `InvalidOperationException` exception, ❶, in this case.
- **Test case 2:** Initialize the `ItemDetail` instance with the wrong item Id. In this case, we will get an `InvalidOperationException` exception, ❷.
- **Test case 3:** Initialize the `ItemDetail` instance with a valid item Id, but the item type as a group. In this case, we will get an `InvalidOperationException` exception, ❸.
- **Test case 4:** Initialize the `ItemDetail` instance with a valid item Id and the item type is an entry, ❹.

We can implement these test cases in an `ItemDetailTests` bUnit test class as shown here in *Listing 11.7*:

Listing 11.7: `ItemDetailTests.cs` (<https://epa.ms/ItemDetailTests11-7>)

```

namespace PassXYZ.Vault.Tests;

[Collection("Serilog collection")]
public class ItemDetailTests : TestContext {
    SerilogFixture serilogFixture;
    Mock<IDataStore<Item>> dataStore;
    public ItemDetailTests(SerilogFixture fixture) {
        serilogFixture = fixture;
        dataStore = new Mock<IDataStore<Item>>();
        Services.AddSingleton<IDataStore<Item>>
            (dataStore.Object);
    }
    [Fact]
    public void Init_Empty_ItemDetail() {
        var ex = Assert.Throws<InvalidOperationException>(

```

```
        () => RenderComponent<ItemDetail>());
    Assert.Equal(
        "ItemDetail: SelectedItemId is null", ex.Message);
}
[Fact]
public void Load_ItemDetail_WithWrongId() {
    var ex = Assert.Throws<InvalidOperationException>(() =>
        RenderComponent<ItemDetail>(parameters =>
            parameters.Add(p => p.SelectedItemId, "Wrong Id")));
    Assert.Equal("ItemDetail: entry cannot be found with
        SelectedItemId", ex.Message);
}
[Fact]
public void Load_ItemDetail_WithGroup() {
    Item testGroup = new PwGroup(true, true) {
        Name = "Default Group",
        Notes = "This is a group in ItemDetailTests."
    };
    datastore.Setup(x => x.GetItem(It.IsAny<string>(),
        It.IsAny<bool>())) .Returns(testGroup);
    var ex = Assert.Throws<InvalidOperationException>(() =>
        RenderComponent<ItemDetail>(parameters =>
            parameters.Add(p => p.SelectedItemId, testGroup.Id)));
    Assert.Equal("ItemDetail: SelectedItemId should not be
        group here.", ex.Message);
}
[Fact]
public void Load_ItemDetail_WithEmptyFieldList() {
    Item testEntry = new PwEntry(true, true) {
        Name = "Default Entry",
        Notes = "This is an entry with empty field list."
    };
    datastore.Setup(x => x.GetItem(It.IsAny<string>(),
        It.IsAny<bool>())) .Returns(testEntry);
    var cut = RenderComponent<ItemDetail>(parameters =>
        parameters.Add(p => p.SelectedItemId, testEntry.Id));
```

```

        cut.Find("article").MarkupMatches(
            $"<article><p>{testEntry.Notes}</p></article>");
    }
}

```

The first test case is implemented in `Init_Empty_ItemDetail`, ③. In the test setup, we just try to render the `ItemDetail` component directly without passing it a selected item Id. We expect an `InvalidOperationException` exception to be thrown.

Before we can run the test case, we need to resolve the `IDataStore` dependency first. `ItemDetail` has a dependency on the `IDataStore<Item>` interface. We can resolve this using dependency injection. In our app, this dependency is registered in `MauiProgram.cs`.

With `bUnit`, dependency injection is supported using `TestContext`. We can register the dependency using `AddSingleton()`, ②. To isolate the test, we use the `Moq` mocking framework, ①, to replace the actual implementation of `IDataStore`, so we can reduce the complexity of the test setup.

Using `Moq`, we only need to fake the method or property that we need in our test setup. It can help to isolate our tests from their dependencies. To use the `Moq` framework, we can create a `Moq` object using the interface or class that we need as a type parameter. Later, we can define the behavior of the target interface or class when we use it. In the constructor, we create a `Mock` object and register the `IDataStore<Item>` interface using `dataStore.Object`:

```

dataStore = new Mock<IDataStore<Item>>();
Services.AddSingleton<IDataStore<Item>>(dataStore.
    Object);

```

After we register `IDataStore` in the constructor, we can execute the first test case again. This time, we can get the exception and verify the message is what we expect:

```

[Fact]
public void Init_Empty_ItemDetail() {
    var ex = Assert.Throws<InvalidOperationException>(
        () => RenderComponent<ItemDetail>());
    Assert.Equal("ItemDetail: SelectedItemId is null",
        ex.Message);
}

```

Next, let us look at the second test case. In the second test case, we pass an invalid Id to `ItemDetail` and try to render it:

```

[Fact]
public void Load_ItemDetail_WithWrongId() {

```

```

var ex = Assert.Throws<InvalidOperationException>(() =>
RenderComponent<ItemDetail>(parameters =>
parameters.Add(
p => p.SelectedItemId, "Wrong Id")));
Assert.Equal("ItemDetail: entry cannot be found with
SelectedItemId", ex.Message);
}

```

In this case, we also get an expected exception, and we can verify its content using `Assert.Equal`.

In the third test case, we pass a valid `Id` to `ItemDetail`, but the item type is a group. This is a case that is hard to repeat in an integration test or user acceptance test. In a unit test, it is quite easy to verify as we can see here:

```

[Fact]
public void Load_ItemDetail_WithGroup() {
    Item testGroup = new PwGroup(true, true) {
        Name = "Default Group",
        Notes = "This is a group in ItemDetailTests."
    };
    dataStore.Setup(x => x.GetItem(It.IsAny<string>(),
        It.IsAny<bool>())) .Returns(testGroup);
    var ex = Assert.Throws<InvalidOperationException>(() =>
RenderComponent<ItemDetail>(parameters =>
parameters.Add(p => p.SelectedItemId, testGroup.Id));
    Assert.Equal("ItemDetail: SelectedItemId should not be
group here.", ex.Message);
}

```

To test it, we need to create a group and assign it to a `testGroup` variable. In this test case, we need to call the `GetItem()` method of `IDataStore`. Since we mocked `IDataStore` in our setup, here, we need to mock the `GetItem()` method as well. The `Moq` method returns `testGroup` when it is called. After the test setup is ready, we can render `ItemDetail` with `testGroup.Id`. The test result is the exception that we expect.

In the final test case, we will pass a valid `Item Id` and the item type is an entry:

```

[Fact]
public void Load_ItemDetail_WithEmptyFieldList() {
    Item testEntry = new PwEntry(true, true) {

```

```
        Name = "Default Entry",
        Notes = "This is an entry with empty field list."
    };
    dataStore.Setup(x => x.GetItem(It.IsAny<string>(),
        It.IsAny<bool>())) .Returns(testEntry);
    var cut = RenderComponent<ItemDetail>(parameters =>
        parameters.Add(p => p.SelectedItemId, testEntry.Id));
    cut.Find("article").MarkupMatches(
        $"<article><p>{testEntry.Notes}</p></article>");
    Debug.WriteLine($" {cut.Markup}");
}
```

This test case is similar to the third test case, except we can create an entry and assign it to `testEntry` variable. After we render `ItemDetail` with `testEntry.Id`, we can verify that the `<article>` rendered HTML tag is the one that we expect.

So far, we have learned how to test Razor components using `bUnit`. We can see that we can achieve a very high level of test coverage using `bUnit`. This is one of the advantages of Blazor UI design.

We have now completed all the topics that we wanted to explore on unit test development with .NET MAUI in this chapter.

Summary

In this chapter, we introduced unit test development for .NET MAUI apps. There are multiple test frameworks available. We chose `xUnit` as the framework in this chapter. In the MVVM pattern, the unit test of the model layer is the same as with any other .NET application. We developed test cases for the `IDataStore` interface to test our model layer. For the unit test of the view and view model, we focused on the eBlazor Hybrid app using the `bUnit` test library. We can develop an end-to-end unit test for a Blazor Hybrid app with the `xUnit` framework and `bUnit` library. With `bUnit`, we covered topics such as Razor templates, the `RenderFragment` delegate, dependency injection, and the `Moq` framework.

Given the knowledge about unit testing in this chapter, you should now be able to work on your own unit test development. Please refer to the *Further reading* section to find more information on .NET unit test development.

Unit testing can be part of a CI/CD pipeline. With the CI/CD setup, we can run unit tests automatically in the development process. We will discuss this topic further in the next chapter.

Further reading

- *Microsoft Visual Studio 2005 Unleashed* by Lars Powers and Mike Snell
- <https://www.amazon.com/Microsoft-Visual-Studio-2005-Unleashed/dp/0672328194>
- MSTest
- <https://github.com/microsoft/testfx>
- *Strengthening Visual Studio Unit Tests* by John Robbins
- <https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/march/bugslayer-strengthening-visual-studio-unit-tests>
- *NUnit Pocket Reference* by Bill Hamilton
- <https://www.amazon.com/NUnit-Pocket-Reference-Running-OReilly/dp/0596007396>
- NUnit Releases at SourceForge
- <https://sourceforge.net/projects/nunit/>
- *Why Did we Build xUnit 1.0*
- <https://xunit.net/docs/why-did-we-build-xunit-1.0>
- xUnit documentation
- <https://xunit.net/>
- xUnit.NET 2.0 release note
- <https://xunit.net/releases/2.0>

Deploying and Publishing in App Stores

After we complete the development work, we want to publish our app in various app stores. Since .NET MAUI is a cross-platform framework, we can build the same source code for Android, iOS, macOS, and Windows. It is possible to deploy our app to a repository such as GitHub, but most users of these platforms use app stores instead. We need to know how to prepare our app for different app stores. This is the topic of this chapter. In this chapter, we will cover the preparation of the application packages before publishing.

We will cover the following topics in this chapter:

- Preparing application packages for publishing
- Automating the build process using GitHub Actions

Technical requirements

To test and debug the source code in this chapter, we need to install Visual Studio 2022 in both Windows and macOS environments. Please refer to the *Development environment setup* section in *Chapter 1, Getting Started with .NET MAUI* for the full details about environment setup.

We will build Windows and Android packages using Windows and build iOS and macOS packages using macOS.

The source code of this chapter is available in the following GitHub repository:

<https://github.com/PacktPublishing/.NET-MAUI-Cross-Platform-Application-Development/tree/main/Chapter12>

Preparing application packages for publishing

In previous chapters, there is very little platform-specific knowledge required in the .NET MAUI development. However, we cannot avoid platform-specific information when we prepare to publish our app to individual app stores. In this chapter, we will introduce what we need to prepare the app for publishing and then we will introduce how to automate the process using **GitHub Actions**.

What to prepare for publishing

To prepare for publishing, we will focus on the work that we need to do before submitting the package to an app store. After the packages are loaded into your chosen app store, please refer to the documents in each app store about the actual publishing process.

In the preparation for publishing, we are trying to answer these questions:

- How to identify an app in app store
- How to identify the app developers
- Which devices can the app support

To build and sign application packages on different platforms, there is platform-specific configuration involved. In .NET MAUI, the platform-specific information is included in the Visual Studio project file and the platform-specific configuration file. In the Visual Studio project file, conditional compilation is used to specify platform-specific information. We can refer to *Table 12.1* for an overview of what we need to change for each platform.

Item	Android	Windows	iOS	macOS
Package format	.apk / .aab	.msix	.ipa	.app / .pkg
Signing	Keystore	signing certificate Package.StoreAssociation.xml	Distribution certificate Distribution profile	
ApplicationId	package = "com.passxyz.vault2"	Publisher = "CN=F81DB4B-AF4A-473E-ADEA-A55EE7432C05"	<key> CFBundleIdentifier </key> <string>com.passxyz.vault2</string>	
ApplicationVersion	android:versionCode = "1"	Version = "0.3.8.0"	<key> CFBundleVersion </key> <string>2</string>	
Configuration file	AndroidManifest.xml	Package.appxmanifest	Info.plist	

Table 12.1: The build configuration

In *Table 12.1*, to identify an app, the `ApplicationId` and `ApplicationVersion` variables are defined in the Visual Studio project file. For each platform, there is a platform configuration file.

When distributing our app for Android, we generate a `.apk` file or a `.aab` file. The `.apk` file is the original Android package format, which can be used to install an app package on the device or emulator. The `.aab` file is used to submit the app to the Google Play Store. Before submission, we need to sign the package using Keystore. `ApplicationId` and `ApplicationVersion` are mapped to the package ID and version code in the Android configuration `AndroidManifest.xml` file.

When distributing our app for iOS or macOS, a `.ipa` file is generated for iOS. A `.app` or `.pkg` file is generated for macOS. To sign an iOS or a macOS package, we need a distribution certificate and a distribution profile. `ApplicationId` is mapped to the bundle id and `ApplicationVersion` is mapped to the bundle version in `Info.plist`.

When distributing our app for Windows, the MSIX package format is used. We will build the package in a `.msix` file extension. In Windows, a **universally unique identifier (UUID)** is used instead of `ApplicationId`. This UUID is generated as `ApplicationGuid` as an application identifier. `ApplicationVersion` is mapped to the `Version` attribute of the `Identity` tag in `Package.appxmanifest`.

What is MSIX?

MSIX is a new Windows app package format that can be used for all Windows apps. Please refer to the Microsoft documentation to find out more information:

<https://learn.microsoft.com/en-us/windows/msix/overview>

In the subsequent sections, we will introduce how to generate release packages on each platform. We will build Windows and Android packages on Windows and, iOS and macOS packages on macOS. We will try to do it using both Visual Studio and the command line.

Publishing to Microsoft Store

We can build a `.msix` package for Microsoft Store using either Visual Studio or the command line on Windows.

In Visual Studio, we need to set the target framework as **net6.0-windows10.0.19041.0** and set the build type to **Release**.

After that, we can right-click on the project node and select the **Publish...** menu item.

As we can see in *Figure 12.1*, a window will appear with **Select distribution method** on it. We can select **Microsoft Store under a new app name** and click on the **Next** button.

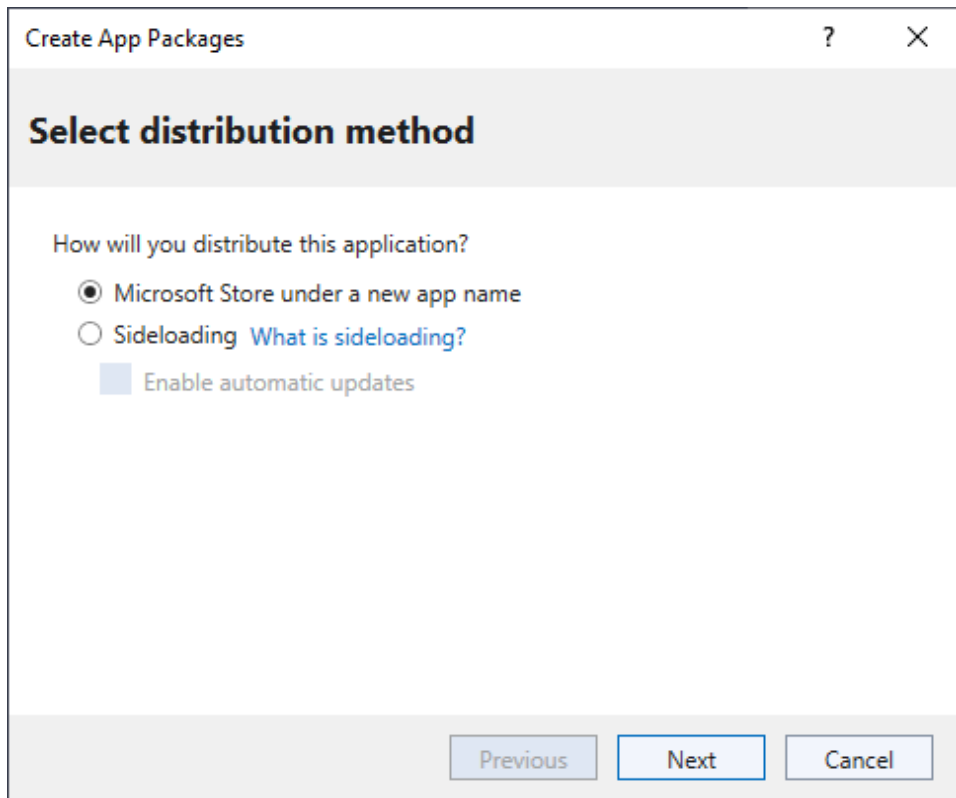


Figure 12.1: Select distribution method

Before we move to the next step in *Figure 12.2*, we need an app name ready.

To create a new app name, we need to do it in Microsoft Store at the following URL:

`http://developer.microsoft.com/dashboard`

Once we have an app name, we can associate it with our app. You can see how to do this in *Figure 12.2*.

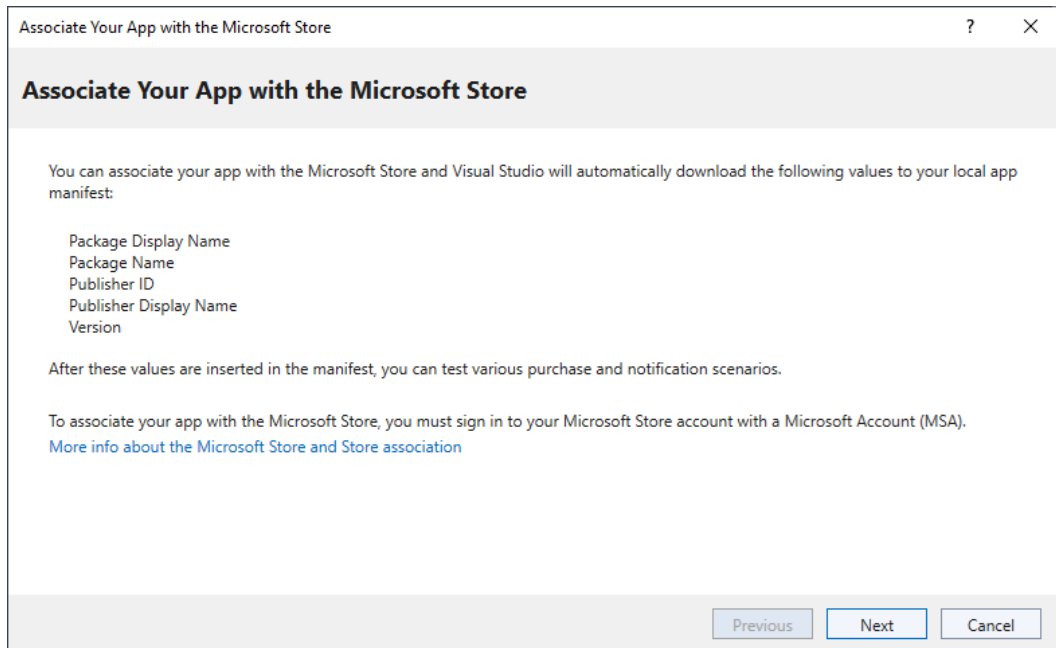
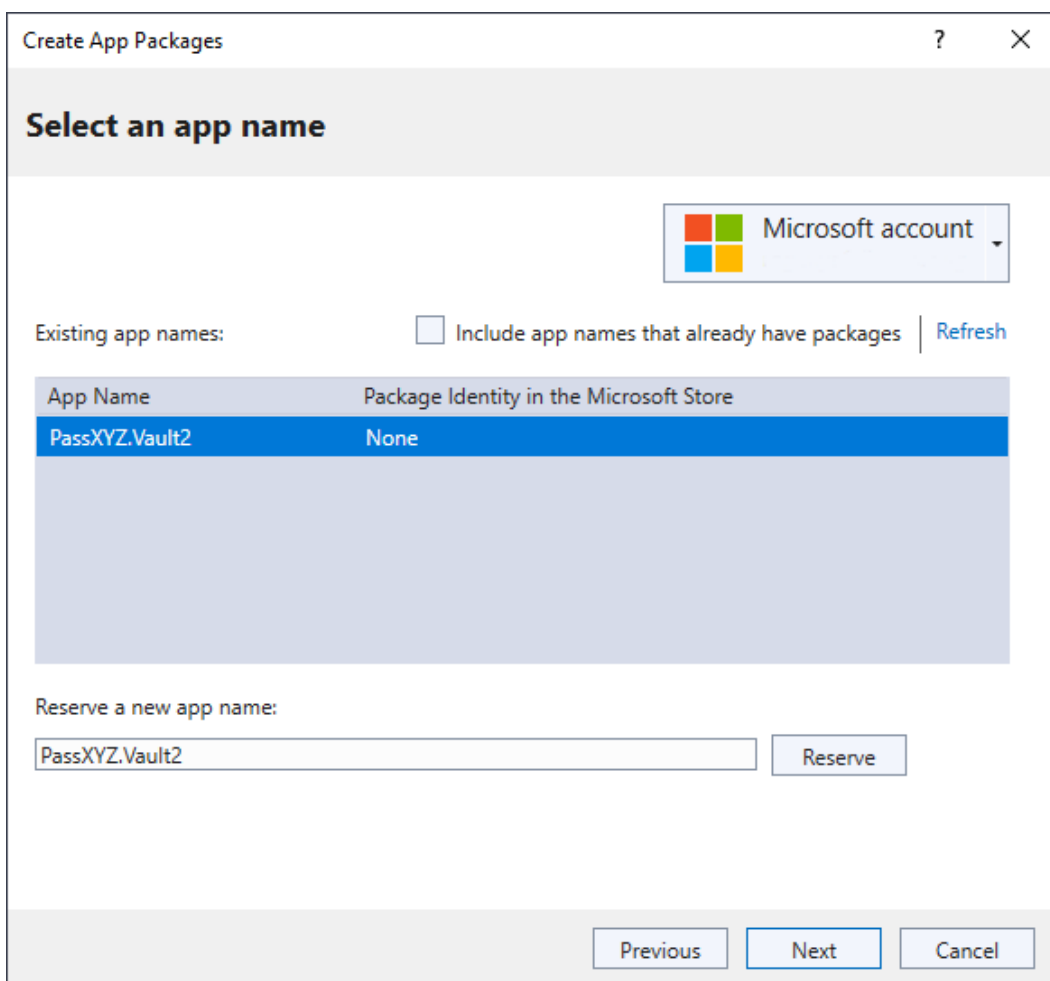


Figure 12.2: Associating your app with Microsoft Store

After we click the **Next** button, Visual Studio will search for the app name in Microsoft Store for us. The app name created in Microsoft Store is shown in *Figure 12.3*.



The screenshot shows a window titled "Create App Packages" with a question mark and a close button in the top right corner. The main heading is "Select an app name". In the top right, there is a Microsoft account selection button with the Microsoft logo and the text "Microsoft account". Below this, the text "Existing app names:" is followed by a checkbox labeled "Include app names that already have packages" and a "Refresh" link. A table lists existing app names and their package identities in the Microsoft Store. The table has two columns: "App Name" and "Package Identity in the Microsoft Store". The first row shows "PassXYZ.Vault2" with a package identity of "None". Below the table, the text "Reserve a new app name:" is followed by a text input field containing "PassXYZ.Vault2" and a "Reserve" button. At the bottom of the window, there are three buttons: "Previous", "Next", and "Cancel".

App Name	Package Identity in the Microsoft Store
PassXYZ.Vault2	None

Figure 12.3: Selecting an app name

After selecting the app name, click on the **Next** button. A screen to select and configure packages will appear, as shown in *Figure 12.4*.

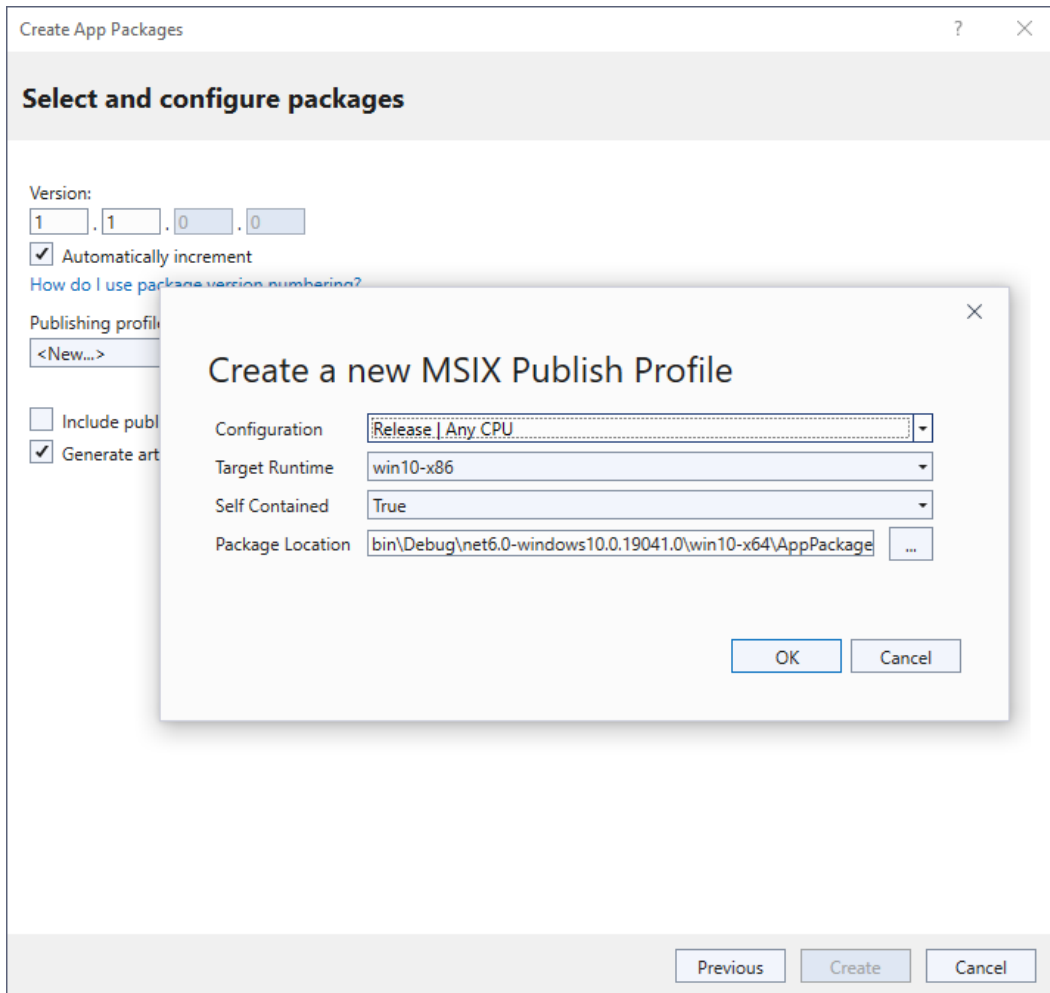


Figure 12.4: Selecting and configuring packages

We can click on the drop-down menu under **Publishing profile**. A dialog will be shown, as we can see in *Figure 12.4*. After clicking the **OK** button in the dialog box, it will create a new MSIX publish profile. Once we have a publish profile, we can click on the **Create** button (which will now become active) to create the package. It will take a while to finish the build and package creation. Once it completes, we can see the following screen, as shown in *Figure 12.5*, and an MSIX package is now ready to be submitted.

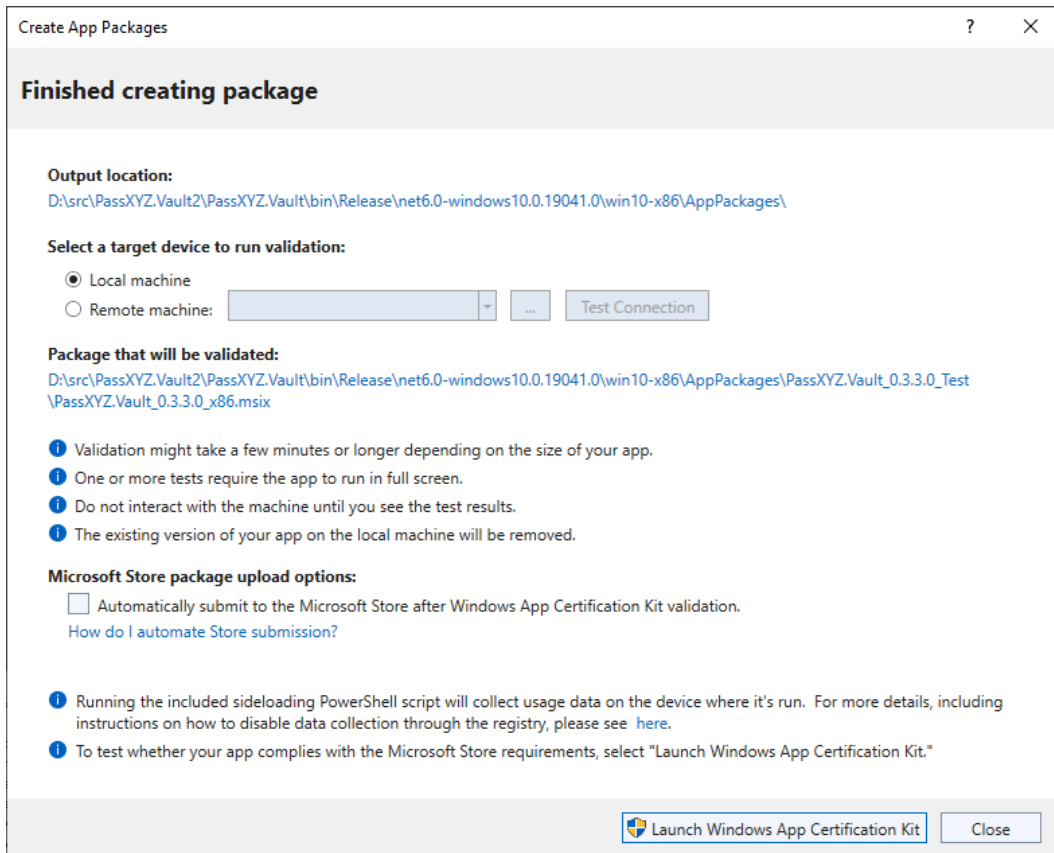


Figure 12.5: An MSIX package

The location of the new package is shown in *Figure 12.5*. There is an option with which we can verify the package by running **Windows App Certification Kit**.

In the preceding steps, two files (as shown here) relating to the publication of the app will have been created in the project folder:

- `Package.StoreAssociation.xml` – this is a file to associate the app with Microsoft Store
- `Properties\PublishProfiles\MSIX-win10-x86.pubxml` – this is the publish profile

Both files may contain sensitive information so they should not be checked into the Git repository.

To integrate the build process in the CI/CD environment, we need to execute the build process using the command line. To build a .msix package using the command line, we can execute the following command from the project folder:

```
dotnet publish PassXYZ.Vault/PassXYZ.Vault.csproj -c Release -f
net6.0-windows10.0.19041.0
```

Once we build the .msix package, we can upload it to Microsoft Store in the **Packages** section of the app submission.

Publishing to the Google Play Store

To prepare for the submission in the Google Play Store, you need to create a new app in the Google Play Console. To create a new app in the Google Play Console, you need a Google account.

To identify an app, every Android app has a unique application ID or package ID defined in the configuration file `AndroidManifest.xml`. This configuration file is generated by Visual Studio from the project file, and it can be found at `Platforms/Android/AndroidManifest.xml`. Let's review `AndroidManifest.xml` of our app in *Listing 12.1*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.passxyz.vault2"
  android:installLocation="auto"
  android:versionCode="1">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/appicon"
    android:roundIcon="@mipmap/appicon_round"
    android:supportsRtl="true"></application>
    <uses-permission
      android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="
      android.permission.INTERNET" />
  </manifest>
```

Listing 12.1: AndroidManifest.xml (<https://epa.ms/AndroidManifest12-1>)

In our app, the application ID is "com.passxyz.vault2" ❶, which is generated from ApplicationId, and the version is the value of android:versionCode ❷, which is generated from ApplicationVersion.

Here is the declaration of the app identifier and version in the PassXYZ.Vault.csproj project file:

```
<!-- App Identifier -->
<ApplicationId>com.passxyz.vault2</ApplicationId>
<ApplicationIdGuid>8606B3B5-C03C-41D7-825F-B33718CF791C
  </ApplicationIdGuid>

<!-- Versions -->
<ApplicationDisplayVersion>1.0</ApplicationDisplayVersion>
<ApplicationVersion>1</ApplicationVersion>
```

To sign an Android package, we need to create a Keystore file. Please refer to the following Android document for information on how to create a Keystore file and sign an Android app:

<https://developer.android.com/studio/publish/app-signing>

Once we have a Keystore file and have prepared the preceding configuration, we need to set the target framework as **net6.0-android** and set the build type as **Release** in Visual Studio.

We can right-click now the project node and select **Publish...** After that, the build will start and we can see an archive has been created, as shown in *Figure 12.6*.

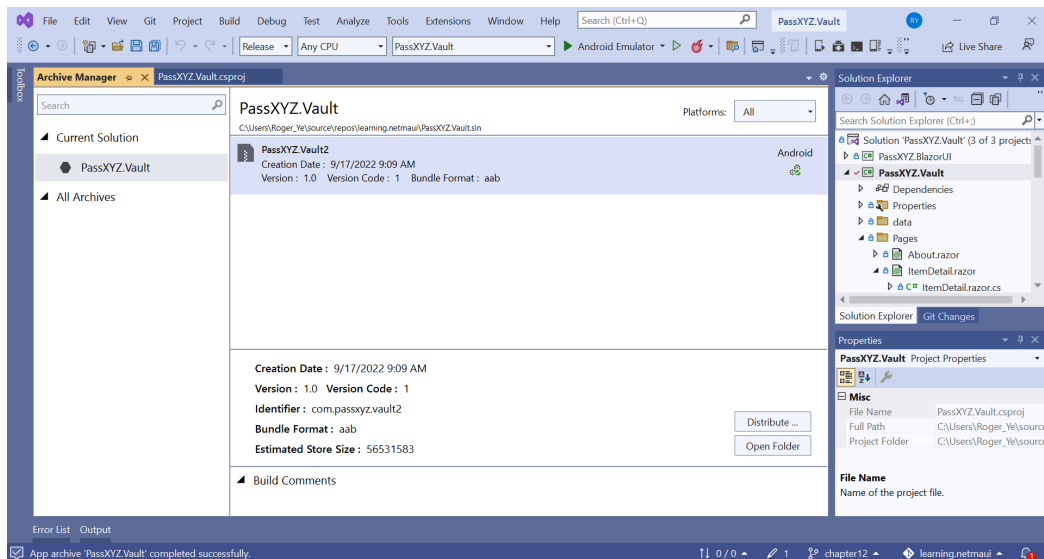


Figure 12.6: Creating an archive for Android

Once the package is created, we can sign it by clicking the **Distribute ...** button, as shown in *Figure 12.6*.

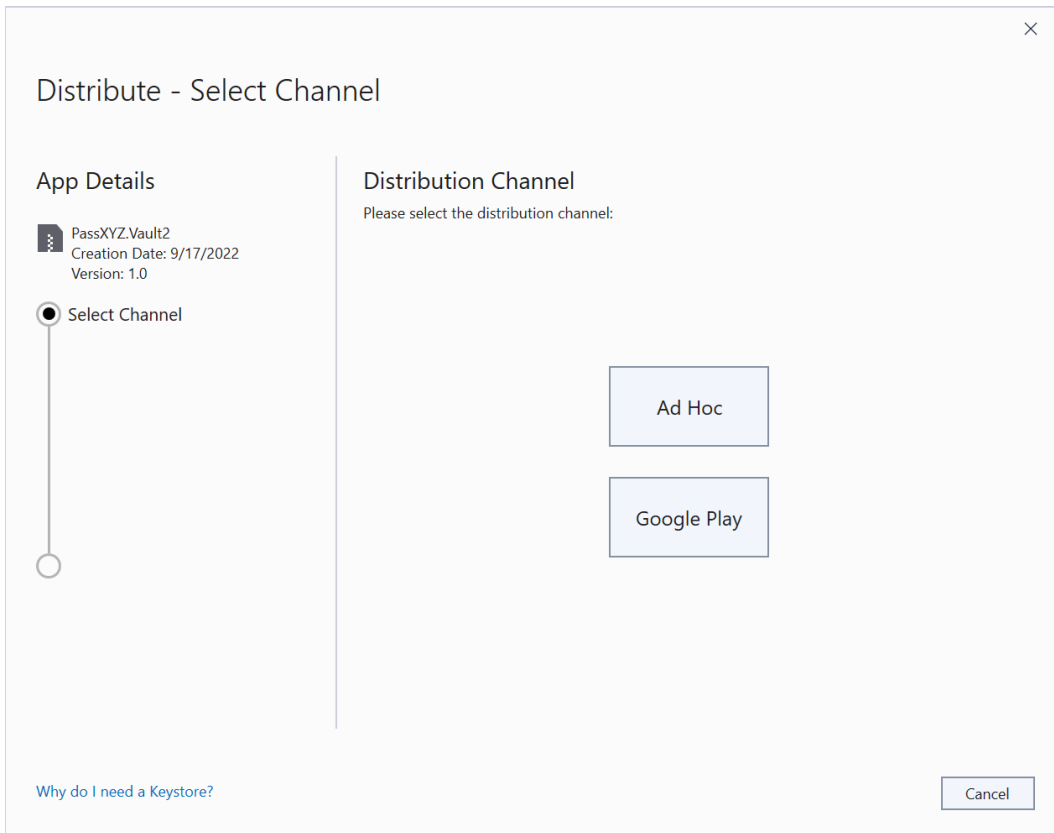


Figure 12.7: Selecting a channel

Once we click the **Distribute ...** button, we need to select a distribution channel, as shown in *Figure 12.7*.

It is possible to sign and submit the package by selecting **Google Play**, but we will select **Ad Hoc** to sign it. We will submit the signed package to Google Play manually later.

Once we select **Ad Hoc**, we can see a different screen as shown in *Figure 12.8*.

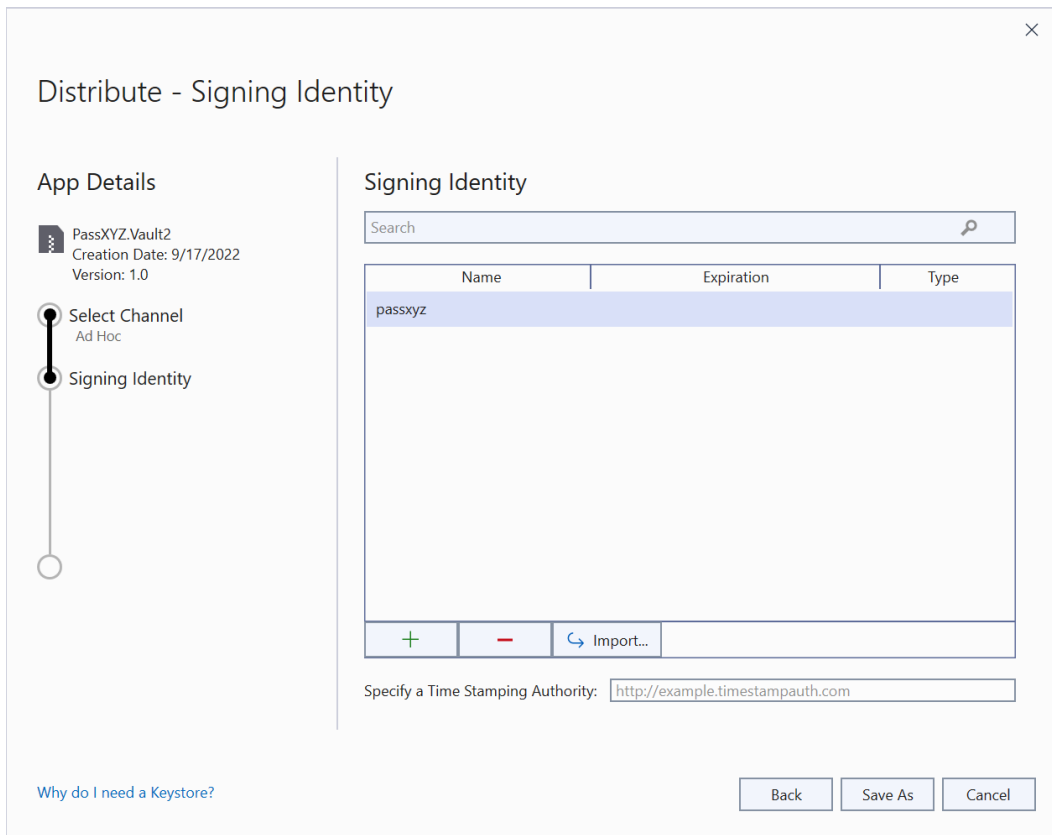


Figure 12.8: Signing Identity using a Keystore file

As shown in *Figure 12.8*, we can click the + button to add a Keystore file. After that, we can click the **Save As** button to sign the package.

The signed .aab file can be submitted to the Google Play Store in the Google Play Console.

If you don't have an existing Keystore file, you may want to follow the guide to creating a new one. The default location of Keystore files is at %USERPROFILE%\AppData\Local\Xamarin\Mono for Android\Keystore\.

To create the package from the command line, we can execute the following command in the project folder:

```
dotnet publish PassXYZ.Vault/PassXYZ.Vault.csproj -c Release -f net6.0-android
```

To learn how to upload a signed Android App Bundle to the Google Play Store, please refer to the following Android document:

<https://developer.android.com/studio/publish/upload-bundle>

Publishing to Apple's App Store

We can introduce the submission of an iOS or macOS app in the App Store together since they have many similarities.

In iOS or macOS apps, bundler identifier and bundler version are used to identify an app. This information is stored in the `Info.plist` configuration file. The bundler identifier is generated from `ApplicationId` and the bundler version is generated from `ApplicationVersion` in the Visual Studio project file.

To sign the package, we need a signing certificate and a provisioning profile. To create a signing certificate and a provisioning profile, we can refer to the following document:

<https://learn.microsoft.com/en-us/dotnet/maui/ios/deployment/provision>

iOS apps can be distributed through the App Store only. The package for the submission is a file with the `.ipa` extension. macOS apps can also be distributed through the App Store, but the package itself can be installed directly.

Even though we can do some of the publishing steps in the Windows environment, we still need to connect to a network-accessible macOS computer. To reduce the complexity, we use a macOS environment for the build of both iOS and macOS apps. Before we build the packages, we need to update the Visual Studio project file to configure our own signing certificate and distribution profile:

```
<PropertyGroup Condition="$ (TargetFramework.Contains('-ios'))
and '$ (Configuration)' == 'Release'">
  <RuntimeIdentifier>ios-arm64</RuntimeIdentifier>
  <CodesignKey>iPhone Distribution: Shugao Ye (W9WL9WPD24)
</CodesignKey>
  <CodesignProvision>passxyz_2022</CodesignProvision>
</PropertyGroup>

<PropertyGroup Condition="$ (TargetFramework.Contains('-
maccatalyst')) and '$ (Configuration)' == 'Release'">
  <CodesignEntitlement>Entitlements.plist
</CodesignEntitlement>
  <CodesignKey>
    3rd Party Mac Developer Application: Shugao Ye
```

```

(W9WL9WPD24)
</CodesignKey>
<CodesignProvision>passxyz.maccatalyst</CodesignProvision>
</PropertyGroup>

```

As we can see, we can use conditional configuration for both iOS and macOS builds. Different signing certificates and distribution profiles are used for iOS and macOS.

In Visual Studio for macOS, we can also configure signing certificates and distribution profiles in project settings for iOS, as shown in *Figure 12.9*. This setting is not supported for macOS yet at the moment, but you may find it is already available when you read this book.

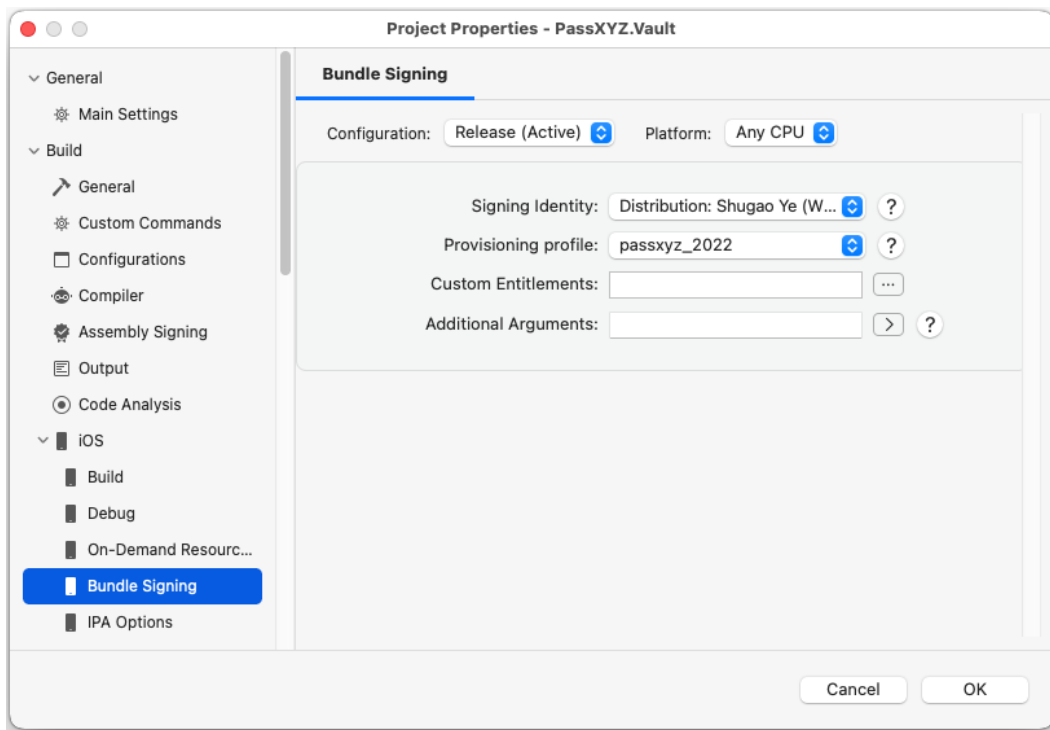


Figure 12.9: The configuration bundle signing

If we are not sure whether the setting is correct or not, we can verify it using **Xcode**. We can create an app in Xcode using the same "com.passxyz.vault2" bundler ID as our app. After that, we can check the configuration of **Signing**, as shown in *Figure 12.10*.

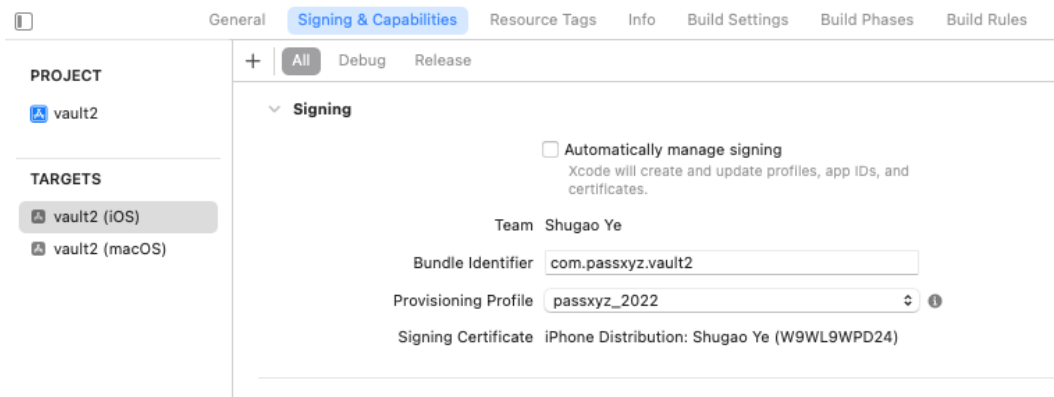


Figure 12.10: The iOS signing settings in Xcode

If there is any issue with the signing certificate or provisioning profile, we can see error messages reported by Xcode. Once the setting is correct in Xcode, the same setting can be used in Visual Studio project without any issues.

With all the configurations ready, we can build the `.ipa` file in the project folder using the following command:

```
dotnet publish PassXYZ.Vault/PassXYZ.Vault.csproj -c Release -f net6.0-ios /p:CreatePackage=true /p:ArchiveOnBuild=True
```

Once the preceding command executes successfully, an `.ipa` file is generated. We can submit this file to the App Store. There are three methods that can be used to upload a package to the App Store. Please refer to the following document to find out more details:

<https://help.apple.com/app-store-connect/#/devb1c185036>

From the preceding document, we know that we can use Xcode, altool, or Transporter to upload a package.

We will use the Transporter app here. After we sign in using the Transporter app, we can upload the package to the App Store, as shown in *Figure 12.11*.

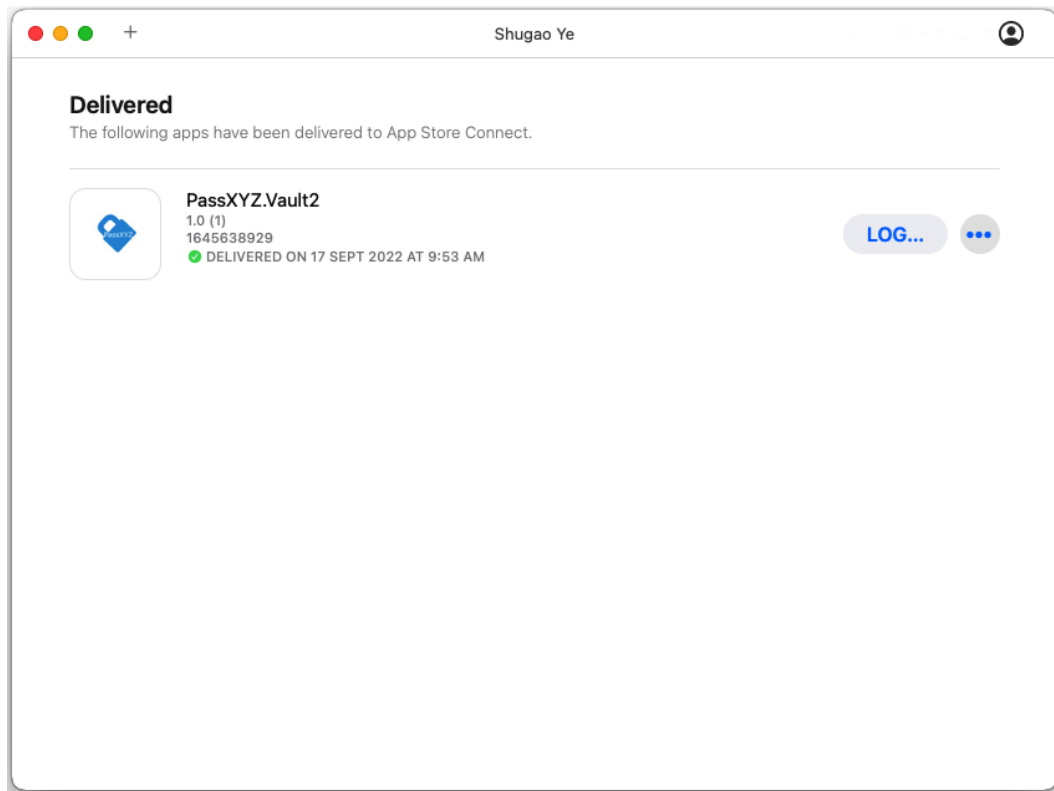


Figure 12.11: Uploading a package using the Transporter app

The building and uploading processes of a macOS package are similar to those for an iOS app. There are three different frameworks (AppKit, MacCatalyst, and SwiftUI) that can be used to build macOS apps. In .NET MAUI, MacCatalyst is used in the platform-specific implementation.

By default, App Sandbox is not enabled in MacCatalyst apps, so we need to enable it. To enable it in the macOS app, we need to add an `Entitlements.plist` file in the build configuration. We can review the `Entitlements.plist` file in *Listing 12.2*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
```

```
<key>com.apple.security.files.user-selected.read-only
</key>
<true/>
<key>com.apple.security.network.client</key>
<true/>
</dict>
</plist>
```

Listing 12.2: Entitlements.plist (<https://epa.ms/Entitlements12-2>)

We cannot verify the configuration of the signing certificate and provisioning profile in Visual Studio for macOS at the moment, but we can verify it in Xcode, as shown in *Figure 12.12*.

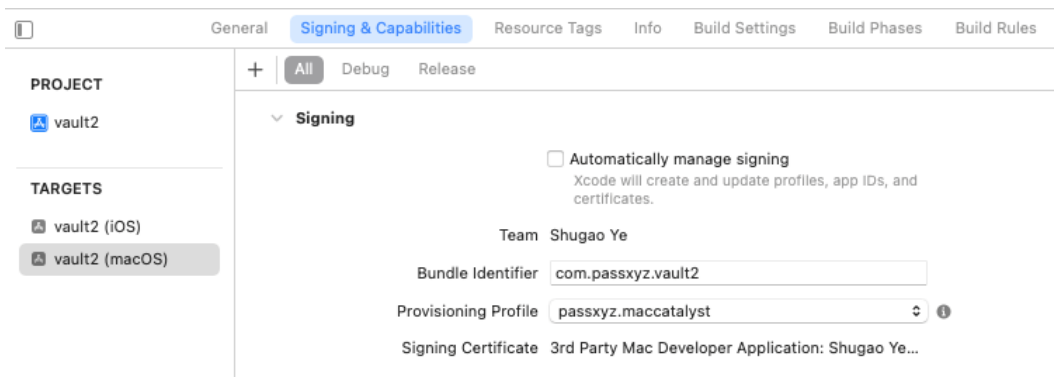


Figure 12.12: The macOS app's Signing settings in Xcode

With all configurations ready, we can build the package in our project folder using the following command:

```
dotnet publish PassXYZ.Vault/PassXYZ.Vault.csproj -c
Release -f net6.0-maccatalyst /p:CreatePackage=true
/p:EnablePackageSigning=true"
```

After we build the package successfully, we can upload the `.pkg` file to the App Store using the Transporter app, as shown in *Figure 12.13*. We can see that we have uploaded both iOS and macOS packages to the App Store successfully.

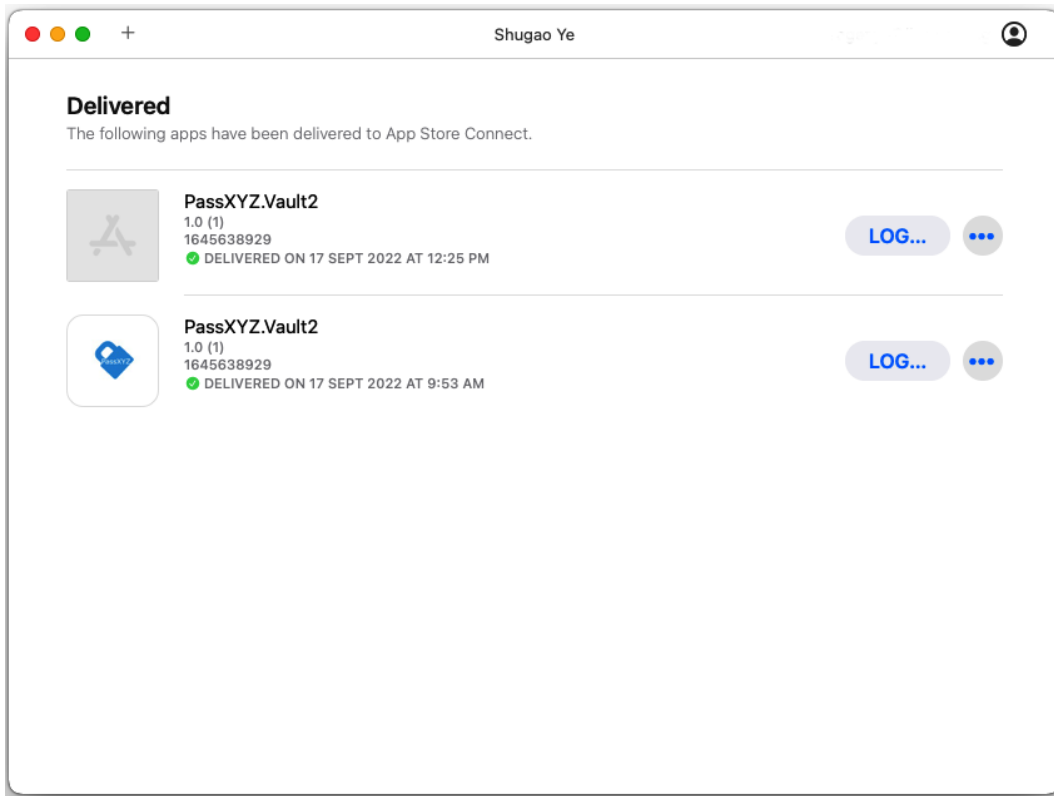


Figure 12.13: Uploading the macOS app using the Transporter app

After we have uploaded packages to Microsoft Store, the Google Play Store, and the App Store, we can test the uploaded packages before the final release using the testing tools provided by the stores:

- The App Store – **TestFlight** can be used to test iOS/macOS apps before the production release
- The Google Play Store – Alpha or Beta testing can be set up before the production release
- Microsoft Store – Package flights can be used on Microsoft Store to test uploaded packages

We have learned the basic steps of how to prepare application packages for supported platforms. With all this in mind, we can explore how to set up the automated build of .NET MAUI app in a **continuous integration and continuous delivery (CI/CD)** environment such as GitHub Actions or Azure DevOps.

GitHub Actions

Since our source code is hosted in GitHub, we will use GitHub Actions as an example to introduce how to set up CI workflows for .NET MAUI development.

Understanding GitHub Actions

GitHub Actions is a CI/CD platform that can be used to support the automation of deployment. For .NET MAUI app development, our target is to build, test, and deploy our apps to app stores or specified publishing channels. In this section, we will focus on CI using GitHub Actions rather than both CI and CD. To deploy apps to various stores, there are many account-specific setup steps, please refer to the .NET MAUI document for the details:

<https://learn.microsoft.com/en-us/dotnet/maui/deployment/>

The GitHub Actions workflow is a process to automatically build and deploy the deliverables from a project. The workflow usually starts from an event such as a `push` or `pull_request` event or when an issue is submitted. Once a workflow is triggered, the defined jobs will start to perform certain tasks inside a runner. Each job consists of one or more steps that either run a script or an action.

In summary, GitHub Actions include events, runners, jobs/steps, actions, and runners.

Workflows

GitHub Actions workflows are defined by a YAML file in the `.github/workflows` directory. YAML is a superset of JSON and it is a better human-readable language. A repository can have one or multiple workflows to perform different jobs. Take a look at *Figure 12.14* to understand the workflow defined in the `PassXYZ.Vault` project.

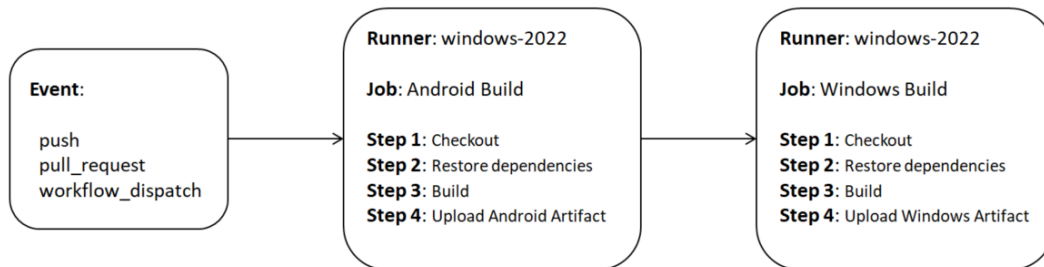


Figure 12.14: The workflow of Windows runner

As we can see in *Figure 12.14*, this is an example of how workflow performs Android and Windows builds. A workflow is triggered by a `push` or `pull_request` event, or manually. It runs inside a Windows runner to perform the builds. When the workflow is triggered, two jobs, **Android Build** and **Windows Build**, will be executed. Each job includes four steps to perform the build, as seen in *Figure 12.14*.

In our project, we defined the two following workflows:

- `passxyz-ci-macos.yml` – This is a workflow to build iOS and macOS on a macOS runner
- `passxyz-ci-windows.yml` – This is a workflow to build Android and Windows on a Windows runner

We can see the YAML files in *Listing 12.3* and *Listing 12.4*:

```
name: PassXYZ.Vault CI Build (Windows)

on:
  push:
    branches: [ master ]
    paths-ignore:
      - '**/*.md'
      - '**/*.gitignore'
      - '**/*.gitattributes'
  pull_request:
    branches: [ master ]
  workflow_dispatch:
permissions:
  contents: read
env:
  DOTNET_NOLOGO: true
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true
  DOTNET_CLI_TELEMETRY_OPTOUT: true
  DOTNETVERSION: 6.0.400
  PROJECT_NAME: PassXYZ.Vault

jobs:
  # MAUI Android Build
  build-android:
    runs-on: windows-2022
    name: Android Build
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Restore Dependencies
```

```
    run: dotnet restore ${env.PROJECT_NAME}}/$
          ${env.PROJECT_NAME}}.csproj
  - name: Build MAUI Android
    run: dotnet publish ${env.PROJECT_NAME}}/$
          ${env.PROJECT_NAME}}.csproj -c Release -f net6.0-
          android --no-restore
  - name: Upload Android Artifact
    uses: actions/upload-artifact@v3
    with:
      name: passxyz-android-ci-build
      path: ${env.PROJECT_NAME}}/bin/Release/net6.0-
            android/*Signed.a*

# MAUI Windows Build
build-windows:
  runs-on: windows-2022
  name: Windows Build
  steps:
    - name: Checkout
      uses: actions/checkout@v3
    - name: Restore Dependencies
      run: dotnet restore ${env.PROJECT_NAME}}
            /${env.PROJECT_NAME}}.csproj
    - name: Build MAUI Windows
      run: dotnet publish ${env.PROJECT_NAME}}/$
            ${env.PROJECT_NAME}}.csproj -c Release -f net6.0-
            windows10.0.19041.0 --no-restore
    - name: Upload Windows Artifact
      uses: actions/upload-artifact@v3
      with:
        name: passxyz-windows-ci-build
        path: ${env.PROJECT_NAME}}/...
```

Listing 12.3: passxyz-ci-windows.yml (<https://epa.ms/passxyz-ci-windows12-3>)

The YAML file in *Listing 12.3* is a little long, but it explains what needs to be set up in a workflow. We will now analyze it step by step.

Events

The workflow is triggered by events that are defined after the `on :` keyword ①. In the preceding workflow, we defined the `push` ②, `pull_request` ③ and `workflow_dispatch` ④ events. For both `push` and `pull_request`, we monitor the events on the master branch. We also ignore no build related commits such as markdown files or configuration files. Please refer to the following GitHub documentation about events that can be used to trigger workflows for more information:

<https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>

Jobs

When a workflow is triggered, it starts to execute the defined jobs. Jobs are defined after the `jobs :` ⑤ keyword. One or more jobs can be defined in a workflow. They are identified by a job ID, such as `build-android` ⑥. There are two jobs, `build-android` and `build-windows`, defined in *Listing 12.3*. Each job can define a name, a runner, and multiple steps.

Runners

A runner is the type of platform that runs the job. In our configuration, both Android and Windows jobs are executed using Windows runners. The runner is defined after the `runs-on :` ⑦ keyword. Please refer to GitHub Actions documentation about the configuration of runners. The runner that we use is `windows-2022`, which is the label of the runner image. In the configuration of the `windows-2022` image, Visual Studio 2022 and .NET MAUI are pre-installed so we can run the build without the installation of any dependencies. However, in the `passxyz-ci-macos.yml` workflow, we need to install .NET MAUI first before we can start the build.

Steps

Multiple steps can be defined in a job and they are defined after the `steps :` ⑧ keyword. In both Android and Windows builds, there are four steps: *checkout*, *restore dependencies*, *build*, and *upload*. Each step can run a script or an action. In the *checkout* step, a `checkout` action is used after the `uses :` ⑨ keyword. An action is a custom application in the GitHub Actions platform to perform a complex but frequent repeated task. Using actions, we can reuse code such as a component in object-oriented programming. To use it, we can just specify the action name with an optional version number. In our script, we can specify the *checkout* action as `actions/checkout@v3`.

The source code of the *checkout* action is hosted on GitHub and can be found at the following site:

<https://github.com/actions/checkout>

In the restore and build steps, we can just run the following **dotnet** command directly after the `run:` syntax:

```
dotnet restore ${env.PROJECT_NAME}/${env.PROJECT_NAME}.csproj
```

After the build is completed, we can upload the artifact using another `upload-artifact` action.

We have introduced the `passxyz-ci-windows.yml` workflow, which performs Android and Windows builds. Let us review the `passxyz-ci-macos.yml` workflow, which performs iOS and macOS builds, in *Listing 12.4*:

```
name: PassXYZ.Vault CI Build (MacOS)

on:
  push:
    branches: [ master ]
    paths-ignore:
      - '**/*.md'
      - '**/*.gitignore'
      - '**/*.gitattributes'
  pull_request:
    branches: [ master ]
  workflow_dispatch:
permissions:
  contents: read
env:
  DOTNET_NOLOGO: true
  DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true
  DOTNET_CLI_TELEMETRY_OPTOUT: true
  DOTNETVERSION: 6.0.400
  PROJECT_NAME: PassXYZ.Vault

jobs:
  # MAUI iOS Build
  build-ios:
    runs-on: macos-12
    name: iOS Build
```

```

steps:
  - name: Checkout
    uses: actions/checkout@v3
  - name: Install .NET MAUI 2
    shell: pwsh
    run: |
      & dotnet nuget locals all --clear
      & dotnet workload install maui --source
      https://aka.ms/dotnet6/nuget/index.json --source
      https://api.nuget.org/v3/index.json
      & dotnet workload install android ios maccatalyst
      tvos macos maui wasm-tools maui-maccatalyst --source
      https://aka.ms/dotnet6/nuget/index.json --source
      https://api.nuget.org/v3/index.json
  - name: Restore Dependencies
    run: dotnet restore ${env.PROJECT_NAME}}
      /${env.PROJECT_NAME}}.csproj
  - name: Build MAUI iOS
    run: dotnet build ${env.PROJECT_NAME}}
      /${env.PROJECT_NAME}}.csproj -c Release -f
      net6.0-ios --no-restore /p:buildForSimulator=True
      /p:packageApp=True /p:ArchiveOnBuild=False
  - name: Upload iOS Artifact
    uses: actions/upload-artifact@v3
    with:
      name: passxyz-ios-ci-build
      path: ${env.PROJECT_NAME}}/bin/Release/net6.0-
        ios/iossimulator-x64/**/*.app

# MAUI MacCatalyst Build
build-mac:
  runs-on: macos-12
  name: MacCatalyst Build
  steps:
    - name: Checkout

```

```
uses: actions/checkout@v3
- name: Install .NET MAUI
  shell: pwsh
  run: |
    & dotnet nuget locals all --clear
    & dotnet workload install maui --source
      https://aka.ms/dotnet6/nuget/index.json --source
      https://api.nuget.org/v3/index.json
    & dotnet workload install android ios maccatalyst
      tvos macos maui wasm-tools maui-maccatalyst --source
      https://aka.ms/dotnet6/nuget/index.json --source
      https://api.nuget.org/v3/index.json
- name: Restore Dependencies
  run: dotnet restore ${env.PROJECT_NAME}
    /${env.PROJECT_NAME}.csproj
- name: Build MAUI MacCatalyst
  run: dotnet publish ${env.PROJECT_NAME}
    /${env.PROJECT_NAME}.csproj -c Release -f
    net6.0-maccatalyst --no-restore -p:BuildIpa=True
- name: Upload MacCatalyst Artifact
  uses: actions/upload-artifact@v3
  with:
    name: passxyz-macos-ci-build
    path: ${env.PROJECT_NAME}/bin/Release/net6.0-
      maccatalyst/maccatalyst-x64/publish/*.pkg
```

Listing 12.4: passxyz-ci-macos.yml (<https://epa.ms/passxyz-ci-macos12-4>)

The workflow of the iOS and macOS build is similar to the workflow of the Android and Windows build. The difference is that the `macos-12` ❶ runner is used here. Visual Studio for macOS is pre-installed in this runner, but `.NET MAUI` is not installed at the moment. We need to add an extra step to install `.NET MAUI` ❷ before the build. The rest of the steps are similar to a Windows or Android build.

We introduced the configuration of all builds in GitHub Actions. Let us check the build status on GitHub.

The screenshot shows the GitHub Actions interface for the repository 'shugaoye / PassXYZ.Vault2'. The workflow 'Updated passxyz-ci-windows.yml' is shown as completed. The summary section indicates the workflow was triggered by a push 18 days ago, with a status of 'Success', a total duration of '9m 50s', and '2' artifacts. The jobs section lists 'Android Build' and 'Windows Build', both marked as successful. The artifacts section shows two artifacts: 'passxyz-android-ci-build' (111 MB) and 'passxyz-windows-ci-build' (50.5 MB).

shugaoye / PassXYZ.Vault2 Public

forked from passxyz/Vault2

Code Pull requests Actions Projects Wiki Security Insights Settings

Updated passxyz-ci-windows.yml PassXYZ.Vault CI Build (Windows) #5 Re-run all jobs

Summary

Jobs

- Android Build
- Windows Build

Triggered via push 18 days ago

shugaoye pushed -> 6261287 master

Status: Success

Total duration: 9m 50s

Artifacts: 2

passxyz-ci-windows.yml

on: push

- Android Build 9m 29s
- Windows Build 2m 24s

Artifacts

Produced during runtime

Name	Size
passxyz-android-ci-build	111 MB
passxyz-windows-ci-build	50.5 MB

Figure 12.15: The Android and Windows build status

From *Figure 12.15*, we can see that both Android and Windows builds are completed successfully. The build artifacts can be downloaded from GitHub after the build.

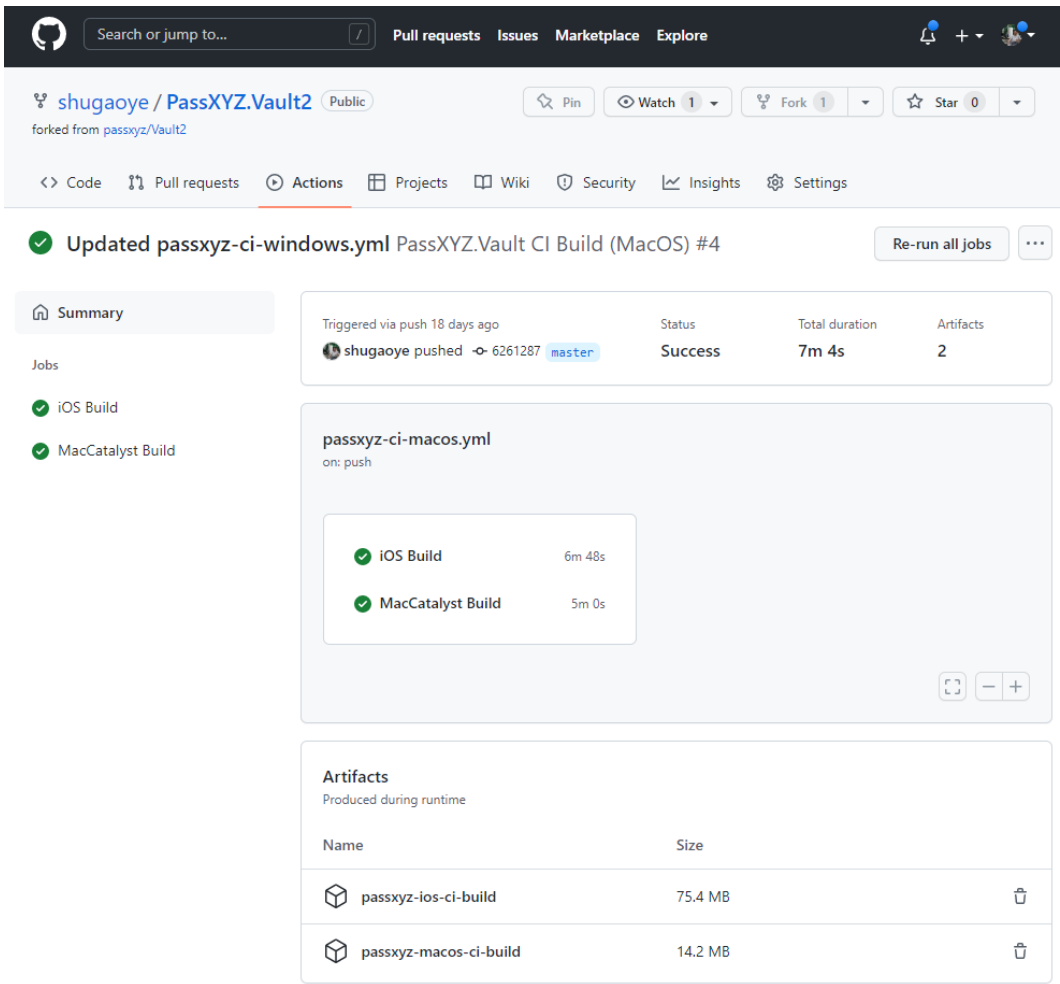


Figure 12.16: The iOS and MacCatalyst build status

In *Figure 12.16*, we can see that both iOS and MacCatalyst builds are completed successfully. With the successful builds in GitHub Actions, we have concluded the introduction of packaging our app for app store submission and automating build using GitHub Actions.

Summary

CI/CD are common practices in today's development process. In this chapter, we introduced how to prepare the build and submit packages for various app stores. The process after the submission of build packages is not covered since they are platform and account-specific topics.

After we introduce the build process of each platform, we can automate the process in GitHub Actions. In the second part of this chapter, we introduced how to set up the build process in GitHub Actions.

With all the skills that you learned from this book, you should be able to develop your own .NET MAUI applications and be ready to submit your apps to supported app stores now.

Index

Symbols

.NET

- versus Java 5-7
- unit test frameworks 304
- versus JavaScript 5-7

.NET Generic Host 27

.NET landscape

- exploring 7

.NET MAUI 12

- architecture 13
- ContentPage 118
- controls 70
- development environment, setting up 15
- Flyoutpage 118
- improvement 13
- installing, on macOS 17, 18, 19
- installing, on Windows 16, 17
- layouts 71
- navigation mechanism 114
- selecting, XAML versus Razor 15
- TabbedPage 118
- used, for developing cross-platform applications 19

.NET MAUI application startup

- platform entry point 156, 157

.NET MAUI Blazor app

- migrating to 190, 191
- startup code 187-190
- versus React Native 182

.NET MAUI Blazor project

- creating 182, 183
- creating, with Visual Studio on Windows 184, 185
- generating, with dotnet command line 183, 184
- running 185, 186

.NET MAUI DI configuration 155-158

.NET MAUI document

- implementation, pros and cons 182
- reference link 353

.NET MAUI project

- app startup and lifecycle management 27-29
- creating, with dotnet command 27
- creating, with Visual Studio 23-26
- setting up 23

.NET Standard 8

A

Ahead-of-Time (AOT) 10

Android build

- debugging 44, 45

Android Studio 182

Apple's App Store

publishing to 347-52

application packages

preparation, for publishing 336, 337

AppName 236

app, signing

reference link 344

app startup and lifecycle management,

.NET MAUI project

lifecycle override methods,

consuming 32-35

states 29, 30

Window lifecycle events,

subscribing to 30, 31

app, uploading Play Console

reference link 347

ASP.NET built-in Razor components

used, for validating data 285, 286

B

backend services 4

Base Class Library (BCL) 6

base modal dialog component

creating 241-243

Binding class

reference link 93

Blazor 15, 176

dependency injection 203, 204

MVVM pattern 201-203

using, advantage 201

Blazor components

versus Razor components 177, 178

Blazor Hybrid

Blazor App 181

exploring 179, 180

Blazor Hybrid app 15, 176, 179

Blazorise

reference link 237

Blazor layout components

using 214-217

Blazor Server 176, 177

Blazor WebAssembly (Wasm) 177, 178

BlazorWebView

setup 208, 209

BootstrapBlazor

reference link 237

built-in MS.DI DI service

constructor injection 158, 159

DI configuration 155-158

lifetime management 153, 154

property injection 159, 160

usage 151, 152

bUnit 305

C

C# 176

camel case 235

cascading values and parameters

communicating with 253-255

child content rendering 247-249

class fixtures

used, for sharing context

between test 312-314

client-side routing 208

BlazorWebView, setup 208, 209

Router, setup 209, 210

collection fixtures

used, for sharing context

between test 315-317

Common Language Infrastructure (CLI) 7

Common Language Runtime (CLR) 6, 66

Common Type System (CTS) 6

component parameters

- ConfirmDialog 246
- defining 244
- ModalDialog 245

constructor

- used, for sharing context
between test 310-312

context menu

- Dropdown component, creating 277-281

continuous integration and continuous delivery (CI/CD) 352**controls, .NET MAUI 70**

- Editor 70
- Entry 71
- Image 70
- Label 70

Create, Read, Update, Delete, and List (CRUDL) operations 135, 136**cross-platform applications**

- developing, with .NET MAUI 19

cross-platform technologies 4, 5

- backend services 4
- languages and frameworks versus
Microsoft solutions 5
- native applications 4
- overview 3
- web application 4

CRUD operations

- fields 266-271
- implementing 262
- item, adding 262, 263
- item, deleting 263-266
- item, editing 263-266

CRUD operations, on database

- item, adding 163-168
- item, deleting 168-171
- item, editing 168-171
- performing 163

CSS styles 204-206**custom font icons**

- displaying 40-43
- setting 38
- setup 38, 39
- using, advantages 38

D**database**

- connecting to 160-162
- CRUD operations, performing 163
- initialization 162

data binding 90-94, 243

- BindingContext property 93
- collections 104-111
- components 92
- notifications, modifying in
viewmodels 96-98
- properties 90
- SetBinding method 93

data binding, modes 94, 95

- OneTime binding 95
- OneWay binding 95
- OneWayToSource binding 95
- TwoWay binding 95

data model

- code efficiency, improving 136
- Command interface 140-142
- enhanced design 134
- improving 98, 99, 133
- KPCLib 99, 100
- login process, improving 137-139
- MVVM pattern 134, 135
- password entries and groups, processing
with IDataStore interface 135
- PassXYZLib 102
- services 134, 135

- updating 103
- use cases 133
- users, processing with IUserService
 - interface 136
- data service**
 - improving 98, 99
 - updating 103, 104
- Data Transfer Objects (DTOs) 88**
- data validation, with ASP.NET**
 - built-in Razor components**
 - built-in components, using 286, 287
 - EditForm component, using 287
 - EditFormDialog component,
 - creating 288-299
- dependency injection (DI) 145, 149**
 - built-in MS.DI DI service, using 151, 152
 - DependencyService 149, 150
 - in Blazor 203, 204
 - using 149
- Dependency Inversion Principle (DIP) 104, 145, 147**
- DependencyService 149**
 - KPCLib package 150
 - PassXYZLib package 150
 - PassXYZ.Vault 150
 - registration 150
 - resolution 150
- design principles 146**
 - Don't Repeat Yourself (DRY) 146
 - Keep It Simple, Stupid (KISS) 146
 - SOLID design principles 147
 - types, exploring 146
 - using 147, 148
 - You Aren't Gonna Need It (YAGNI) 147
- Document Object Model (DOM) 176**
- Don't Repeat Yourself (DRY) 146**
- dotnet command**
 - .NET MAUI project, creating with 27

- dotnet command line**
 - used, for generating .NET MAUI Blazor project 183, 184
- Dropdown component**
 - creating, for context menu 277-281

E

- EditForm component**
 - using 287, 288
- EditFormDialog component**
 - creating 288-299
- Extensible Application Markup Language (XAML) 11, 15**

F

- flyout 119**
 - items 120
 - menu items 120-122

G

- GitHub Actions 352**
 - events 356
 - jobs 356
 - runner 356
 - steps 356-361
 - workflows 353, 355
- Google Play Store**
 - publishing to 343-347

H

- Havit.Blazor**
 - reference link 237
- hierarchical navigation 114**

I**IDataStore interface** 135

test cases, creating to test 307-309

INavigation interface 114**Interface Segregation Principle (ISP)** 147**Intermediate Language (IL)** 177**iOS**

debugging 45, 46

J**Java**

versus JavaScript 5-7

versus .NET 5, 6, 7

Java Virtual Machine (JVM) 5**JUnit** 304**Just-in-Time (JIT)** 10**K****KeePass** 19**KeePass database format, use cases**

AboutPage 99

ItemDetailPage 99

ItemsPage 99

LoginPage 98

NewItemPage 99

KeePassLib 99**Keep It Simple, Stupid (KISS)** 146**KPCLib** 99**L****languages and frameworks**

versus Microsoft solutions 5

layout

applying, to component 218, 219

nesting 220

layouts, .NET MAUI 71

AbsoluteLayout 74

FlexLayout 73

Grid 72

StackLayout 71, 72

lifetime management, MS.DI 153, 154**Liskov Substitution Principle (LSP)** 147**list view**

implementing 222-228

ListView component

creating 282-284

using 284, 285

localization 81

.resx file, creating 81, 82

of text 83-85

M**macOS**

.NET MAUI, installing 17-19

debugging 45, 46

MainLayout 218**master-detail UI design**

ItemDetailPage 74-76

ItemsPage 76-80

markup extensions 67, 68

navigation 74

side-by-side approach 69

stacked approach 69, 70

MauiApp 157**MauiAppBuilder** 157**MauiProgram** 157**Microsoft solutions**

versus languages and frameworks 5

Microsoft Store

publishing to 337-343

mixed-mode AOT compilation 177

mobile development

Xamarin, using 9, 10

modal navigation 114**Model-View-Controller (MVC) pattern 88**

versus Model-View-ViewModel
(MVVM) pattern 89

Model-View-Presenter (MVP) 88**Model-View-Update (MVU) 13****Model-View-ViewModel (MVVM)**

pattern 87, 88

in Blazor 201, 202, 203

in PassXYZ.Vault 89, 90

versus Model-View-Controller
(MVC) pattern 89

Model-View-ViewModel project

scaffolding 47, 48

Shell template, reusing from
Xamarin.Forms 49-54

Visual Studio project template 54-56

Mono 7, 8

MS.DI DI service. *See* **built-in
MS.DI DI service**

MSIX

reference link 337

N

native application 4**Navbar component**

creating 274-277

navigation elements

Add button, adding 228-230

Back button, adding 228-230

implementing 220, 221

list view, implementing 222-228

navigation implementation

hierarchical navigation 114

INavigation interface 114

modal navigation 114

NavigationPage 115

navigation stack, manipulating 116

navigation stack, using 115, 116

NavigationManager

navigating with 213, 214

NavigationPage 115**navigation stack**

manipulating 116

page, inserting 116

page, removing 117

nested component 246, 247

child content rendering 247-249

NET Framework 7**NuGet package 237****NUnit 304**

O

object-oriented programming (OOP) 146

OnAfterRenderAsync method 258, 260

OnAfterRender method 258-261

OnInitializedAsync method 257

OnInitialized method 257

OnParametersSetAsync method 257

OnParametersSet method 257

Open/Closed Principle (OCP) 147

Open Iconic icons 226

P

PageLayout 220

pascal case 235

password database 160

password manager app

building 19

PassXYZLib 102

PassXYZ.Vault 19
Plain Old CLR Objects (POCOs) 88
pop action 115
portable class libraries (PCLs) 8
Progressive Web App (PWA) 178
publishing
 preparation 336, 337
push action 115
PwEntry
 abstracting 101
 properties 101
PwGroup
 abstracting 101
 properties 101

R

Razor class library 237
 creating 237-239
 static assets, using 239
Razor component lifecycle 255, 256
 methods 256
 OnAfterRenderAsync method 258-260
 OnAfterRender method 258-261
 OnInitializedAsync method 257
 OnInitialized method 257
 OnParametersSetAsync method 257
 OnParametersSet method 257
 SetParametersAsync method 256
 ShouldRender method 258
Razor components 177, 234, 235
 creating 194, 274
 inheritance 236
 using, to redesign login page 194-201
 versus Blazor components 177, 178
Razor component, with bUnit
 project configuration, modifying 318
 Razor pages, testing 327-332

RenderFragment delegate, using 323-326
 test case, creating 319, 320
 test cases, creating in Razor files 320-323
 testing 317

Razor files

test cases, creating 320-323

Razor Syntax 191

code block 191
 directive attributes 193
 directives 193
 explicit Razor expressions 192
 expression, encoding 192
 implicit Razor expressions 191

Razor templates 320, 322

React Native

cons 182
 platform-specific implementation 181, 182
 pros 182
 versus .NET MAUI Blazor 182

remote procedure calls (RPCs) 176

RenderFragment delegate

using 323-326

resources configuration 36

app icon 37
 custom font icons, setting 38
 splash screen 37

reusable Razor components

base modal dialog component,
 creating 241-243
 cascading values and parameters,
 communicating with 253-255
 component parameters, defining 244
 creating 239-241
 data binding 243
 nested component 246, 247
 two-way data binding 250

route parameters

using, to pass data 211, 213

Router

- setup 209, 210

routes

- defining 210, 211

S**ServiceCollection configuration**

- AddScoped method 153
- AddSingleton method 153
- AddTransient method 153

SetParametersAsync method 256, 260**Shell**

- features 118
- flyout 119
- tabs 122
- using 117, 118

Shell navigation 125

- absolute routes, registering 125
- data, passing to pages 127-132
- relative routes, registering 125-127

ShouldRender method 258**SignalR 176****single-page application (SPA) 4****Single Responsibility Principle (SRP) 147****snake case 235****SOLID design principles 147**

- Dependency Inversion Principle (DIP) 147
- Interface Segregation Principle (ISP) 147
- Liskov Substitution Principle (LSP) 147
- Open/Closed Principle (OCP) 147
- Single Responsibility Principle (SRP) 147

StackLayout 71, 72

- HorizontalStackLayout 72
- VerticalStackLayout 72

static assets

- using, in Razor class library 239

T**tabs 122**

- creating 124
- using 122, 123

templated components

- using 281, 282

TestFlight 352**title bar**

- UI elements 229

two-way data binding 250

- component parameters, binding with 250-253

U**unit testing, in .NET 304**

- context, sharing between tests 309
- context, sharing with class fixtures 312-314
- context, sharing with collection fixtures 315-317
- context, sharing with constructor 310- 312
- test cases, creating to test IDataStore interface 307-309
- unit test project, setting up 305-307

UpdateFieldAsync() event handler

- key parameter 270
- value parameter 270

user interface design

- comparing, on platforms 11, 12

V**Visual Studio**

- .NET MAUI project, creating with 23-26
- used, for creating .NET MAUI Blazor project on Windows 184, 185

Visual Studio Community

- download link 16

W

web application 4

WebAssembly (Wasm) 178

Windows

.NET MAUI Blazor project, creating
with Visual Studio 184, 185

.NET MAUI, installing 16, 17

Windows build

debugging 44

Windows Runtime (WinRT) 8

X

Xamarin

using, for mobile development 9, 10

Xamarin.Essentials 10

examples of functionalities 11

Xamarin.Forms 10

features 12

Shell template, reusing from 49-54

XAML 191

markup extensions 66

XAML page

creating 58, 59

XAML syntax 60

attribute syntax 62

element syntax 60, 61

XAML namespaces 62-66

XML namespaces 62-66

Xcode 182, 348

Ximian 7

xUnit 304

Y

You Aren't Gonna Need It (YAGNI) 147



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

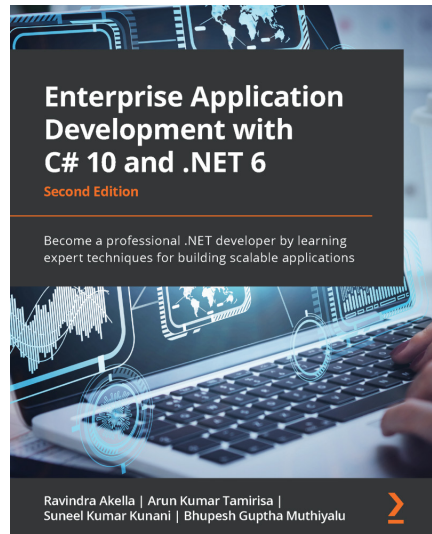
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

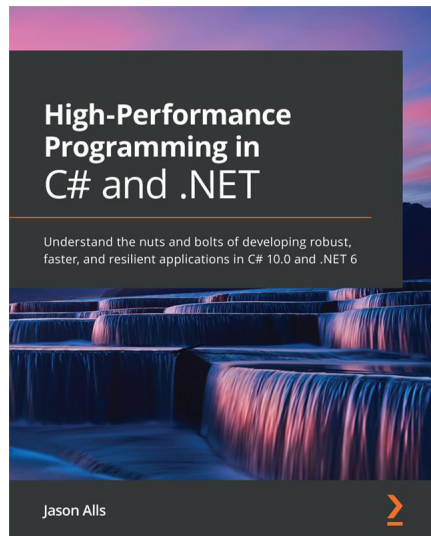


Enterprise Application Development with C# 10 and .NET 6

Suneel Kumar Kunani | Ravindra Akella | Arun Kumar Tamirisa | Bhupesh Guptha Muthiyalu

ISBN: 978-1-80323-297-3

- Design enterprise apps by making the most of the latest features of .NET 6
- Discover different layers of an app, such as the data layer, API layer, and web layer
- Explore end-to-end architecture by implementing an enterprise web app using .NET and C# 10 and deploying it on Azure
- Focus on the core concepts of web application development and implement them in .NET 6



High-Performance Programming in C# and .NET

Jason Alls

ISBN: 978-1-80056-471-8

- Use correct types and collections to enhance application performance
- Profile, benchmark, and identify performance issues with the codebase
- Explore how to best perform queries on LINQ to improve an application's performance
- Effectively utilize a number of CPUs and cores through asynchronous programming
- Build responsive user interfaces with WinForms, WPF, MAUI, and WinUI.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Hi!

I am Roger Ye, author of *.NET MAUI Cross-Platform Application Development*. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency.

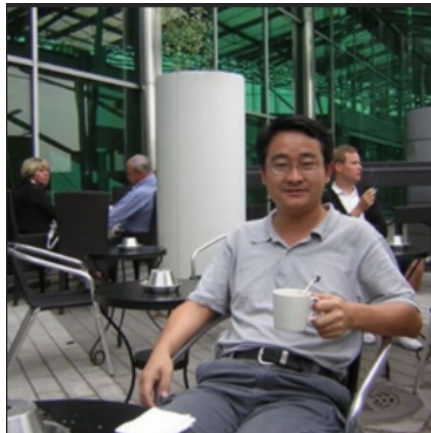
It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on this book.

Go to the link below to leave your review:

<https://packt.link/r/180056922X>

Your review will help us to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best wishes



Roger Ye

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?
Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781800569225>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

