Enterprise Applications with C# and .NET

Develop robust, secure, and scalable applications using .NET and C#

Alexandre F. Malavasi Cardoso

bpb

Enterprise Applications with C# and .NET

Develop robust, secure, and scalable applications using .NET and C#

Alexandre F. Malavasi Cardoso

Enterprise Applications with C# and .NET

Develop robust, secure, and scalable applications using .NET and C#

Alexandre F. Malavasi Cardoso

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-373-1

www.bpbonline.com

Dedicated to

My beloved wife:
Paula

&

My Daughter ***Myla***

About the Author

Alexandre F. Malavasi Cardoso has been working in software development for more than 16 years, playing central roles in numerous projects as a technical leader and software engineer, delivering projects using Microsoft Technologies for big companies, including well succeed projects in South America, Europe, and the United States. Currently, he is a Head of Engineering at Propylon and a Technical Advisor at Marelo companies. He is also an accomplished postgraduate completing a degree in Business and Systems Analysis and holds two master's degrees focused on Software Engineering with Agile Methods Emphasis. In the meantime, he successfully got many Microsoft certifications in Azure and Web Development technologies. Furthermore, the author participates as a speaker in international IT Conferences and writes technical articles on Web Development and related topics. Based on all his contributions to technical communities worldwide, he was awarded three times by Microsoft as Most Valuable Professional (MVP).

About the Reviewers

Gourav, a Developer with more than a decade of experience in developing end to end solution and more than 6 years of experience in front end tech stack. Specialized in web-based solutions having good knowledge of front-end technologies as well. Passionate about learning new technologies, design patterns, using modern technologies in solutions.

He is a good mentor, who has assisted more than 20+ students/juniors in learning more about web design and applications. He loves to share his knowledge via blogs, video tutorials, courses, and so on.

Interested in developing web based solutions using Angular/React and .net core.

Srikeerti is a .NET full stack developer with a passion for creating innovative and scalable applications. With a strong background in computer science and a dedication to staying current with the latest technologies, Srikeerti is skilled in a variety of programming languages, including C#, ASP.NET, and JavaScript. She has experience developing and maintaining complex web applications, working on both the front-end and back-end of development projects. Srikeerti is a strong problem-solver and enjoys the challenges that come with working on large-scale projects. In addition to her technical skills, Srikeerti is a team player and excellent communicator, able to effectively collaborate with cross-functional teams and translate complex technical concepts to a non-technical audience.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my wife Paula and my daughter Myla.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Building enterprise applications is a complex task that requires a comprehensive understanding of the latest technologies and programming languages. C# and .NET are powerful tools that have become increasingly popular in the field of enterprise development.

This book is designed to provide a comprehensive guide to building enterprise applications with C# and .NET. It covers a wide range of topics, including the basics of C# programming, advanced concepts such as object-oriented programming, and the use of the .NET platform for building robust and scalable applications.

Throughout the book, you will learn about the key features of C# and .NET and how to use them to build enterprise applications that are efficient, reliable, and easy to maintain. You will also learn about best practices and design patterns for building enterprise applications and will be provided with numerous practical examples to help you understand the concepts.

This book is intended for developers who are new to C# and .NET and want to learn how to build enterprise applications. It is also helpful for experienced developers who want to expand their knowledge of these technologies and improve their skills in building robust and reliable applications.

With this book, you will gain the knowledge and skills to become a proficient developer in the field of enterprise development using C# and .NET. I hope you will find this book informative and helpful.

Chapter 1: Introduction to .NET- explains everything needed for the reader to develop applications based on the .NET platform and C# language, including detailed instructions on setting up local environments and available tools to build and debug applications. Furthermore, the chapter also gives the reader an overview of the .NET capabilities for multi-platform development and walks through the most common project types available in Visual Studio, including templates for web, desktop, and mobile development.

Chapter 2: Status of the .NET Platform- presents a detailed overview of the history of the .NET platform and shows the differences between different versions of the framework over time, including explanations of the evolution from the .NET Framework to .NET Core versions and the consolidation of the .NET platform until .NET 7. This is essential content for the entire book as this chapter covers fundamental aspects of the .NET platform that influences technical decisions to build enterprise applications and to understand important migration challenges between legacy versions of .NET and newer versions.

Chapter 3: Cross-platform Applications- covers the cross-platform characteristics of the .NET platform, including details on the most recent changes in the compatibility of different types of .NET project types for multiple operating systems and practical examples of generating Asp.Net Core web applications that can be executed in Windows, Linux and macOS operating systems. Furthermore, the chapter shows how to build self-contained executables for Console and Desktop applications.

Chapter 4: The Object-Oriented Programming- allows the reader to learn fundamental concepts related to the Object-Oriented Programming paradigm using C# language, including good practices of software development to develop stable, readable, and extensible code for robust enterprise applications. Furthermore, the chapter explains SOLID principles with details and practical examples and gives recommendations on using inheritance, static classes, structs, and interfaces.

Chapter 5: Interfaces and Inheritance in C#- gives special attention to inheritance and interfaces in C# language, demonstrating how to implement robust and extensible applications and explaining concepts that help the reader solve real complex problems in terms of design and definition of classes in C# language through practical examples based on real scenarios.

Chapter 6: Basic Concepts of Design Patterns- shows basic concepts of Design Patterns and provides practical examples for .NET projects based in C# language to the reader. Learning Design Patterns allows the reader to apply essential features present in .NET combined with the Object-Oriented Programming paradigm. The chapter includes practical examples of Single, Façade, Adapter, Observer, Builder, and Factory patterns.

Chapter 7: Operations, Loops, and Iterations in C#- explains with details and numerous practical examples how to do primary operations

using C# language, including operators, loops, and iterations. This chapter also allows the reader to learn the basics of C# language, including explanations of object types, manipulation of arrays and lists, switch statements, and much more.

Chapter 8: Error Handling and Exceptions in C#- is dedicated to error handling and exceptions in C# to give the reader more familiarity with the main exception types and ways to prevent unexpected errors that can occur in .NET applications. This chapter covers practical examples of working with try-catch blocks and recommends applying robust error-handling strategies when implementing basic programs in C#.

Chapter 9: Using and Understanding LINQ- contains multiple practical examples of using LINQ in C# applications and explains how developers can create expressions, filters, and manipulate data in C# language for objects based on List and Enumerable types.

Chapter 10: Unit Tests- covers creating, structuring, and applying unit tests for programs based in C# language, including numerous practical examples using the xUnit tool and concepts of Tet-Driven Development (TDD). After reading this chapter, developers should be able to code unit tests for legacy and new .NET projects using the xUnit tool and understand the benefits and advantages of having a significant test coverage rate for enterprise applications.

Chapter 11: New Features in C# 8.0 and 9.0- gives the reader opportunity to learn the new features introduced in C# 8.0 and 9.0, including new possibilities for switch expressions, default methods for interfaces, async capabilities for each statement, advanced features to work with indices for arrays, init operator for classes, records, and much more.

Chapter 12: Building .NET Applications for Linux- explains and demonstrates how to build .NET applications for Linux operating systems using Visual Studio and Windows Sub-System for Linux (WSL), with practical examples. The chapter also covers the benefits of cross-platform development for project planning, including advantages for the hiring process of development teams, market opportunities for cross-platform development, and much more.

Chapter 13: Asp.Net Core Web API- covers how to build Asp.Net Core Web APIs using C# language, including a detailed explanation of the nature

of HTTP requests, general API projects' characteristics, and the correct use of HTTP Verbs. This chapter also shows the reader how to build and configure minimal APIs in .NET.

Chapter 14: Blazor, the Single Page Application of .NET- shows the reader a detailed guide for the Blazor Framework, the Single Page Application (SPA) project of the .NET platform for web development using Web Assembly and C# language. Single Page Application became one of the most popular patterns used to build modern and high-performance applications, traditionally using JavaScript as the primary language. The .NET platform has introduced its own SPA, being possible to develop powerful applications using C# language as the primary language, combined with Web Assembly.

Chapter 15: Desktop, Console, and Mobile Applications- introduces the reader to the possibilities that the .NET platform provides for desktop, console, and mobile applications, including details on how to correctly set up the local environment for multi-platform development, with numerous practical examples using the most common project types in Visual Studio.

Chapter 16: Azure Integration Services- covers essential aspects and possibilities for integrating .NET applications with existing cloud services on Azure. Modern software development involves knowledge of cloud services architecture patterns suitable for distributed systems, allowing developers to create scalable, reliable, and cost-effective applications in terms of infrastructure. The .NET platform offers a wide range of libraries and packages to facilitate integration with Azure cloud services. For this reason, this chapter gives the reader practical examples of working with Azure Functions, Azure Storage, and other cloud resources.

Chapter 17: Authentication in Asp.Net Core- explains and demonstrates authentication concepts for Asp.Net Core applications in general, including Web APIs and Authentication for Blazor apps. Security is one of the main aspects of any web application these days. Data confidentiality and trust are key elements for any successful business with software development as it is core. Given that, this chapter prepares the reader to apply authorization and authentication concepts in a recommended way for Asp.Net Core applications.

Chapter 18: Introduction to Entity Framework Core- introduces the reader to the Entity Framework Core, with numerous examples of

integrating .NET applications with databases, including demonstrations of performing CRUD operations in basic applications. This chapter also covers details on Object Relational Mapping (ORM) concepts in general and shows how to use LINQ queries in combination with Entity Framework Core.

Chapter 19: Good Practices for .NET Applications- covers good practices that can be applied to any .NET project, including dependency injection, logging, exception handling, and performance enhancements. After studying this chapter, the reader should be able to understand how to monitor production environments using logging, how to apply dependency injection in .NET applications, how to plan an excellent strategy to create reliable applications with a recommended approach for exception handling, and much more.

Chapter 20: Architecture Concepts for .NET Applications- explains concepts of software architecture that can be applied to .NET applications to develop robust applications and face the challenge of defining the extensible and reliable architecture for enterprise applications, including particular decisions around the platform, DevOps, microservices, cloud architecture, and design patterns.

Chapter 21: Creating an Enterprise Application in .NET- has a complete hands-on project approach that allows the reader to apply all the concepts learned throughout the book. The chapter follows a step-by-step approach, showing how to build a Blazor application using Entity Framework Core, Authentication, and much more.

Code Bundle and Coloured Images

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

https://rebrand.ly/2mwg9vr

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

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

Errata

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

errata@bpbonline.com

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

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.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 BPB books and eBooks.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

CHAPTER 1

Introduction to .NET

Introduction

With this chapter, we are starting our journey into the cross-platform and modern .NET. The .NET platform has changed since 2016 to provide powerful libraries that allow us to build applications for any operating system and even multiple devices, taking the benefits of the best software development practices.

In this chapter, you will learn how to set up your local environment to build .NET applications using Visual Studio or Visual Studio Code, and you will also have the opportunity to get familiar with basic concepts of multi-platform development for Linux, macOS, and other operating systems.

Structure

In this chapter, we will discuss the following topics:

- Tools and environment setup
- Installing Visual Studio
- Installing Visual Studio Code
- Introduction to Visual Studio
- Introduction to Visual Studio Code
- Understanding Multi-Platform concepts
- Overview of project types on the .NET platform

Objectives

After studying this unit, you should be able to install and set up the Visual Studio IDE and Visual Studio Code, understand multi-platform concepts and discuss the available .NET project types.

Tools and environment setup

To get started with software development in .NET and C#, you must install the most recent version of Visual Studio, a complete **Integrated Development Environment** (**IDE**) for creating, compiling, and building your .NET projects. Visual Studio is available for Windows and macOS, both of which are available in the Community Edition for studying. The tool can be downloaded from the official Visual Studio Website.

Furthermore, Microsoft has provided Visual Studio Code, an alternative light version of the editor for .NET and C# applications, which is available not only for Windows and macOS but also for various Linux distributions. Considering this editor is an open-source extensible project, the technical community, IT professionals, and companies around the globe have created tons of free extensions for different languages apart from C# itself. Therefore, it is a suitable tool for cross-platform applications without compatibility concerns. The Visual Studio code can be downloaded from the official Website for free.

Installing Visual Studio

After downloading Visual Studio on the official Website, you must take the following steps for the installation:

1. Double click on the downloaded executable file. Ensure your user on the operating system has permission to install the software.
2. Choose the desired workloads to be installed and set up together with the Visual Studio. For the examples of this book, the following workloads must be installed:
    - Asp.Net and Web development
    - Azure development
    - .NET Desktop development
    - Universal Windows Platform development
    - .NET cross-platform development
    - Mobile development with .NET

    The necessary workloads for the development of all the code samples along with this book are presented in *Figure 1.1*:

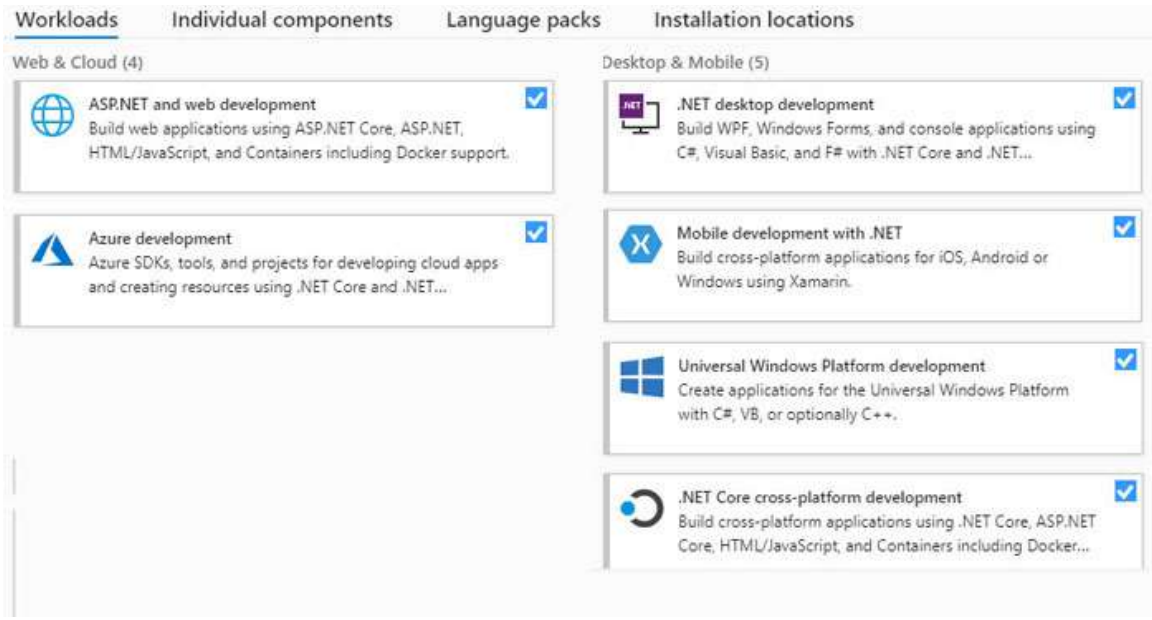


Figure 1.1: *Visual Studio workloads*

3. After choosing the necessary workloads, click on the install option.
4. Usually, Visual Studio is configured to use the same language as the operating system language. If you would like to set up a different one, you can do that in the Language Packs option, where other ones will be available. For the examples of this book, the Visual Studio was configured to the English language.
5. After finishing the installation, you can already create .NET projects.

Note: Once Visual Studio is installed, you don't need to install the latest stable version of .NET SDK, as it is already part of the Visual Studio installation. By default, the Visual Studio updates follow the newest features introduced into .NET, such as library updates, minor and significant changes, and new project templates. Even though the library updates do not automatically apply to your existing projects, each project targets a specific .NET version, and the Visual Studio updates modify just the IDE and not the project configurations.

Installing Visual Studio Code

Visual Studio Code is a cross-platform alternative to Visual Studio IDE, and it is a good option if you want a lightweight editor for .NET projects. The companies and communities have provided extensions that allow us to work with many distinct languages, and it has become one of the most popular editors for software

developers. Also, it is available for any operating system, such as Linux, macOS, and Windows. After downloading the executable from the official Website, you must take the following steps:

1. Double click on the downloaded executable file. Ensure your user in the operating system has permission to install the software.
2. Download the Visual Studio extension for C# and Azure, as shown in *Figure 1.2*:

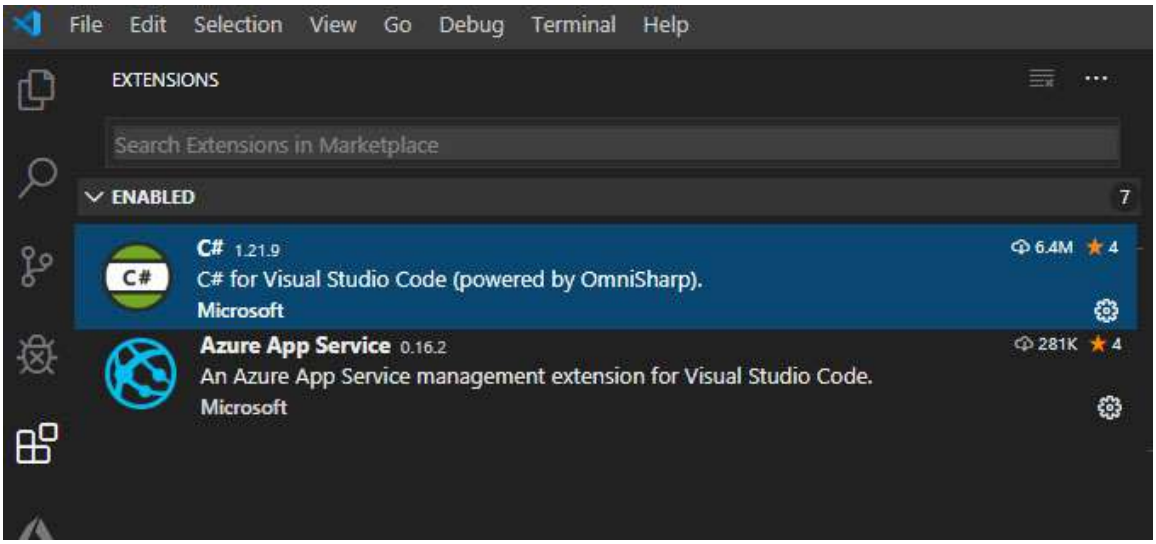


Figure 1.2: *Visual Studio Code extensions*

You must have an Internet connection to search for the extensions on the Extensions tab. After typing the extension name, just choose the install option to complete the process.

Introduction to Visual Studio

Visual Studio is a powerful integrated development environment that allows you to create, build, debug, and deploy your .NET applications in one place, including access to external resources such as databases and Azure features. Also, it contains many project templates to get started with software development, including project types based on .NET. To access those templates, just click on the File option on the superior menu and choose the options New Project, as shown in *Figure 1.3*:

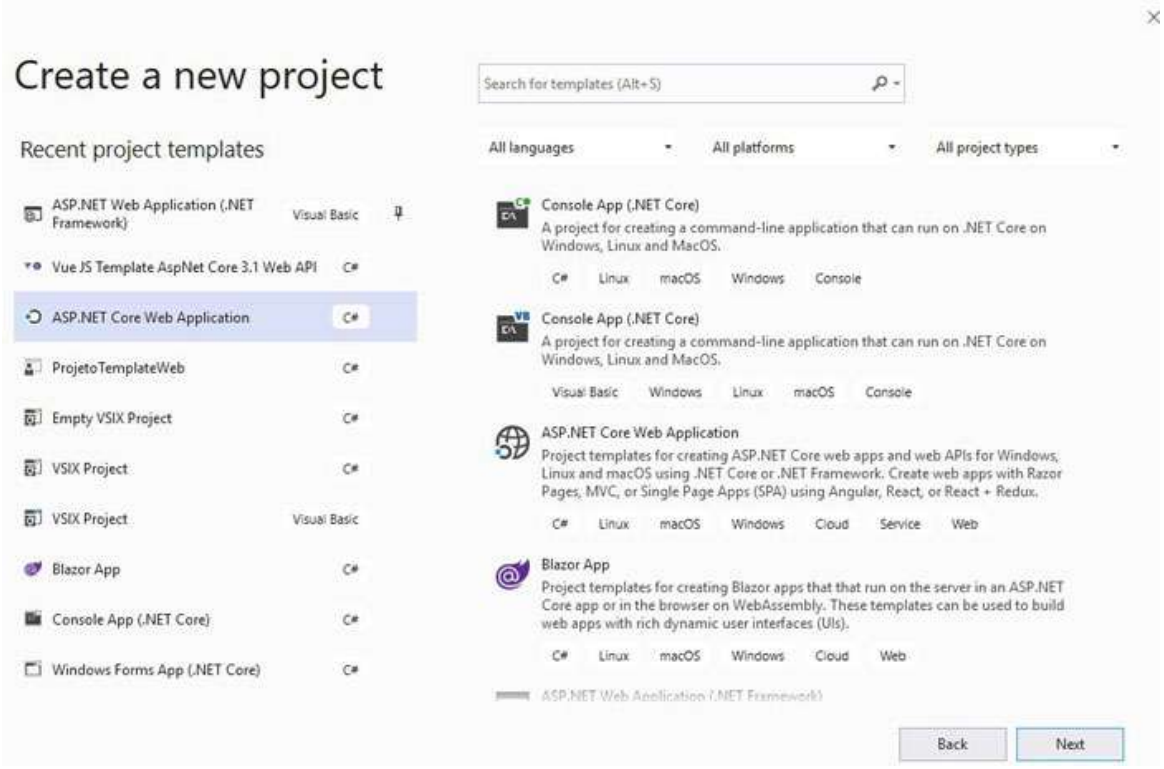


Figure 1.3: *Visual Studio Code extensions*

As seen in *Figure 1.3*, the most popular project templates for .NET, such as the Console App application, Asp.Net Core Web API Application, and Blazor App, are available after installing Visual Studio. Each project template has other sub-types to choose from. Those templates are time-saving and help to configure and create new projects using Visual Studio.

Tip: Open-source communities, companies, and individual developers share many free extra templates on the Visual Studio Marketplace website. Also, Visual Studio extensions for many purposes can be downloaded on the same Website to get the best experience in software development and get integrations with third-party tools. They are also available for Visual Studio and Azure DevOps.

After creating a simple project from the template list, you are redirected to the integrated environment for developing your code, having in a common place the text editor with code suggestions support and access to files and external resources, as shown in *Figure 1.4*:

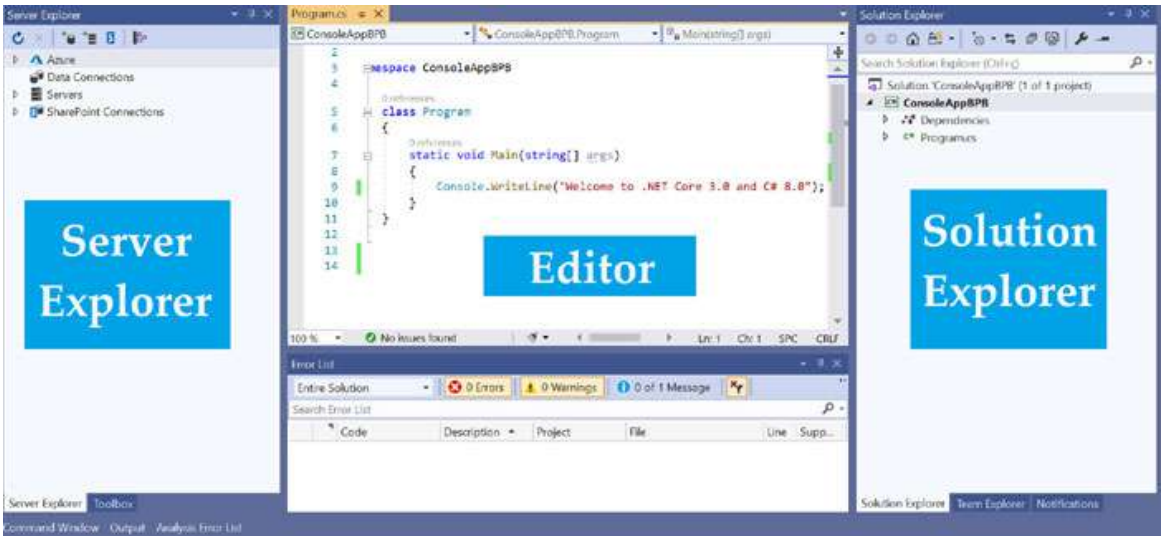


Figure 1.4: *Visual Studio features*

You can access your solution's existing folders and files on the Solution Explorer sidebar. Also, it is possible to see all the linked projects in case you have multiple ones as a part of the software development. As seen in *Figure 1.4*, the **`Editor`** sidebar is where you develop your code. It can get tips on code syntax, indentation, code suggestions, previous codification error messages, and navigate into classes, functions, and methods. The editor is customizable, and many settings, such as background color, contrast, and font size, can be changed.

Finally, on the Server Explorer sidebar, it is possible to connect to external resources, databases, servers, cloud resources, and on-premise features. That way, you can keep all the work in a unique and shared place, which has great value in getting high productivity.

In the **`Toolbar`**, represented in *Figure 1.5*, there are options to run the application, save pending changes, modify the debug mode, comment code lines, open new files, and undo recently added code in the editor:

Figure 1.5: *Visual Studio toolbar*

In the next section, you will learn about Visual Studio Code, a multi-platform IDE option to develop .NET applications.

Introduction to Visual Studio Code

Visual Studio Code seems to be quite different from Visual Studio 2019, but both of them have the exact same purpose: create, debug, build, and deploy applications made using many languages and resources. At its core, it contains many extensions to simplify our routine as developers. The main difference between Visual Studio 2019 and Visual Studio Code is regarding cross-platform development, once Visual Studio Code is available not only in Windows and macOS operation systems, but it is also ready to use in Linux. Another relevant aspect is the performance of this editor compared to a fully integrated development environment. On the other hand, you must enable individual extensions and components according to what you need to develop your software. *Figure 1.6* shows how the main screen of Visual Studio Code is configured:

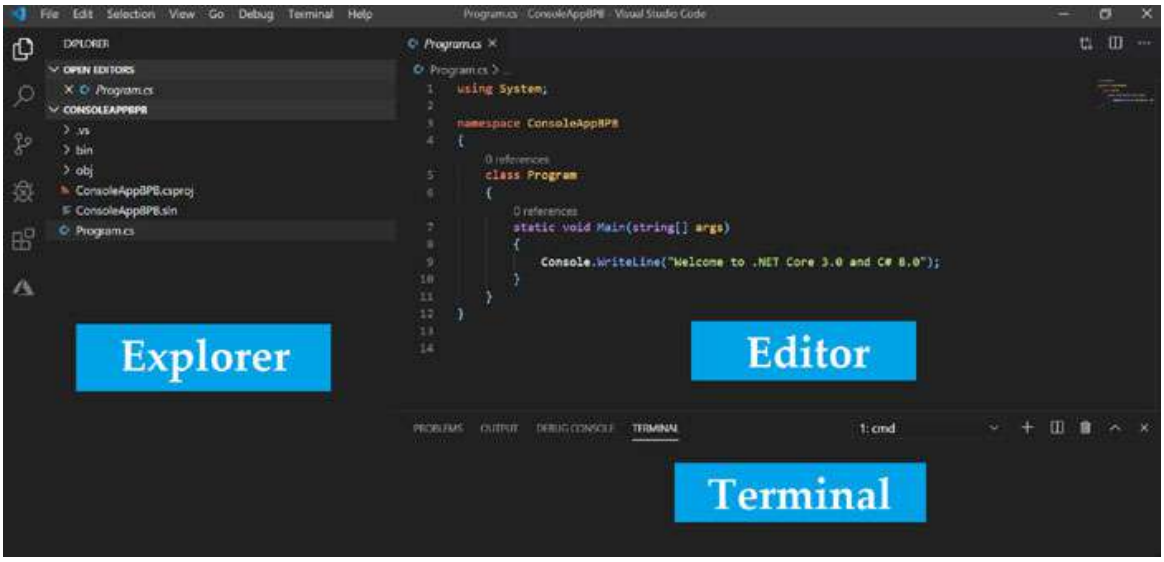


Figure 1.6: *Visual Studio Code*

You can access all your project's existing folders and files on the Explorer sidebar. Also, it is possible to see all the linked projects in case you have multiple ones as part of the software development process. As seen in *Figure 1.5*, the **`Editor`** sidebar is the place where you develop your code and can get tips on code syntax, indentation, code suggestions, previous codification error messages, and navigate into classes, functions, and methods. The editor is customizable, and many settings, such as background color, contrast, and font size, can be changed. Considering you have installed the C# language extension, the editor is ready to highlight the most important parts of your underlying code, showing in distinct color classes, methods, primitive types, and static text such as String, Integer, or even the custom classes you will create. Different extensions were released as open-source projects for other programming languages, but for all the examples in this book, the C# extension is enough to get the desired results.

The Visual Studio Code is a command-line-based editor. Once you are familiar with the commands, you can benefit from quickly running your application and building and creating new files without using mouse clicks. It is essential to be productive and manage your development environment better. All the commands can be executed on the Terminal sidebar, and the output window is shown the results and the status of the commands. Because of the facility to use the editor combined with command lines by a terminal, Visual Studio became the most popular software editor in the world, and it has been well received even by developers who do not use .NET Core and C# for software development. Considering the editor is a cross-platform and open-source project, it has been broadly used by Linux users and for software development in various programming languages and platforms.

Furthermore, Visual Studio Code supports deployment and source control and is easily integrated with GitHub, Gitlab, and Azure DevOps.

Note: Visual Studio was the IDE used to create, run and build all the code examples of this book, but you can get the same result similarly using Visual Studio, using the command lines present in the appendices of this book.

The traditional and complete Visual Studio 2019 IDE version depends on the existent specification in files whose extension is .sln (solution file) and specific extensions for projects such as .csproj for C# library projects. Although Visual Studio Code allows you to run code by only opening simple folders and files, which is helpful in case you want just to run other languages apart from Microsoft technologies.

Visual Studio Code is more flexible regarding productivity and menu options. It contains many pre-defined commands to get files, search across files and contents, debug and deployment options, and manage and add new extensions, as shown in *Figure 1.7*:

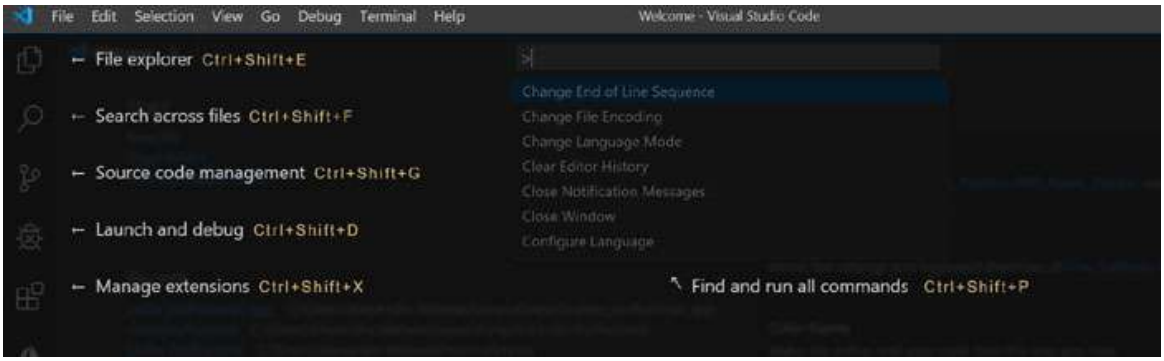


Figure 1.7: *Visual Studio Code - User Interface*

In the next section, you will be presented with multi-platform concepts related to the ability to develop applications for multiple operating systems, devices, and cloud providers.

Understanding multi-platform concepts

With the beginning of cloud-based applications in the last decade, the market competition significantly changed from a battle for a software license to cloud system infrastructure services to support the high demand for scalability, modern applications, big data, globalization, and an open market. Therefore, the interest in profitability is no longer focused on operating systems but on reliable infrastructure, as many companies are changing their business model to provide solutions based on the **Software as a Service (SaaS**) concept.

Nonetheless, big IT companies are still releasing and creating new operation systems, the attention is entirely on open-source projects nowadays, and any modern application must be able to run on Linux, macOS, and Windows platforms. Furthermore, with the rise of the **Internet of Things** (**IoT**), the types of devices where applications can be run are much more diverse, making the software development process more complex. Not long ago, every Web developer's biggest concern was successfully running the same application in multiple different browsers properly, which still represents a considerable challenge. But, although web applications are accessed from multiple operating systems, usually, the application is hosted in a server whose specifications are under control, using a single operating system and platform. This scenario rapidly moved to another more complex one, based in a cross-platform environment to have more compatibility, decrease costs, and deliver applications faster to more relevant users across the globe, using a micro-service architecture.

Therefore, to meet these new requirements, Microsoft provided the .NET Core platform. This rich and powerful technology was made to run everywhere and in any operation system, rewriting all the libraries presented on the .NET Framework and getting better results in terms of performance. At the very beginning of the C# language and .NET Framework, they were made entirely to support only the Windows platform, which was an explicit limitation for adoption in the market because in specific scenarios, such as building Desktop applications, the projects of this type require to be written twice or even more times redundantly and, therefore, increase the costs of any project.

Building applications using .NET Core allows you to write a unique application that can be hosted on any platform in many types of devices and distinct cloud

infrastructures and to consume it in plenty of Web APIs. It represented a revolutionary change for the .NET world and placed this technology in a prominent position in the market. Also, another important movement of .NET was the fact that it became an open-source project. So, individuals, companies, and academic institutions could contribute to the platform's evolution, and since it became true, the .NET community has been one of the most active communities in the world.

Overview of principal project types in .NET

To get started with .NET Core, we must be aware of the main type of projects it supports. Along with the examples in this book, you will have the opportunity to create various types of applications for multiple platforms (Web, Desktop, and mobile). This section only represents a brief overview of all the types used for building enterprise applications. By default, Visual Studio already contains specific project templates to facilitate the creation of .NET Core projects. Some of them are provided by Microsoft, and others by the technical community as open-source projects.

Once you start a new project using Visual Studio, the following options will be available, as shown in *Figure 1.8*:

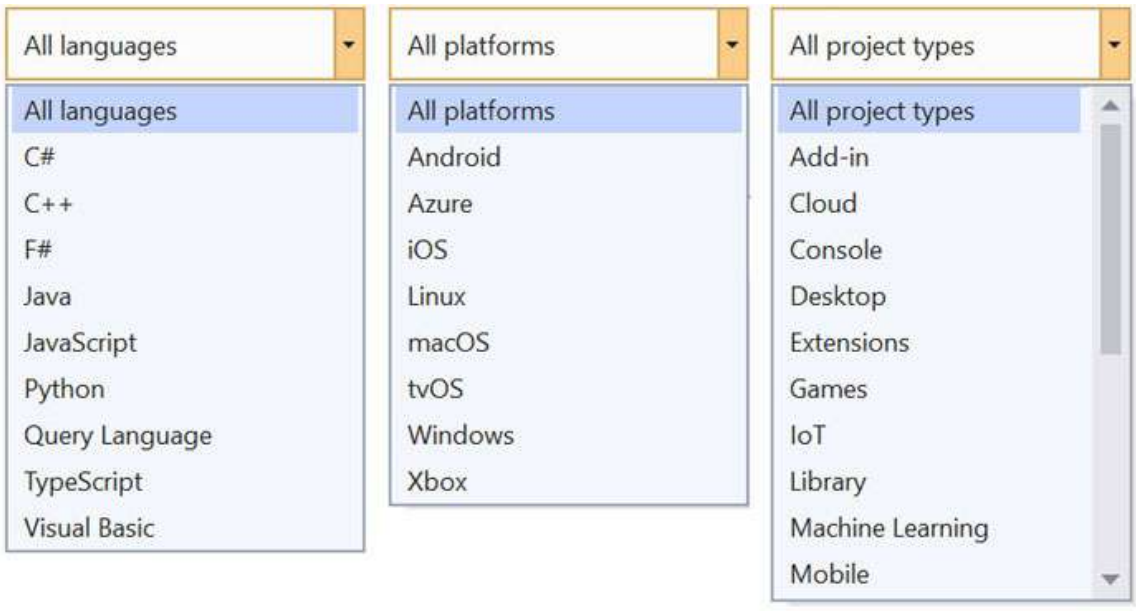


Figure 1.8: *Visual Studio Project options*

As you can see, a wide variety of programming languages, platforms, and project types are available for .NET applications, including mobile app development, desktop applications, Web development, machine learning, and games.

To get started with the .NET project types, we will create a simple **Console App**, a standard project for desktop applications that allows us to run applications in the background and integrate them with simple user interaction. After opening Visual Studio and accessing the project type list (*Figure 1.8*), you must choose the C# language and Windows platform to create the project. It will filter the results to show only the suitable types. Additionally, you can type the word “Console” in the search box. That way, you will get the **Console application** template types, as shown in *Figure 1.9*:

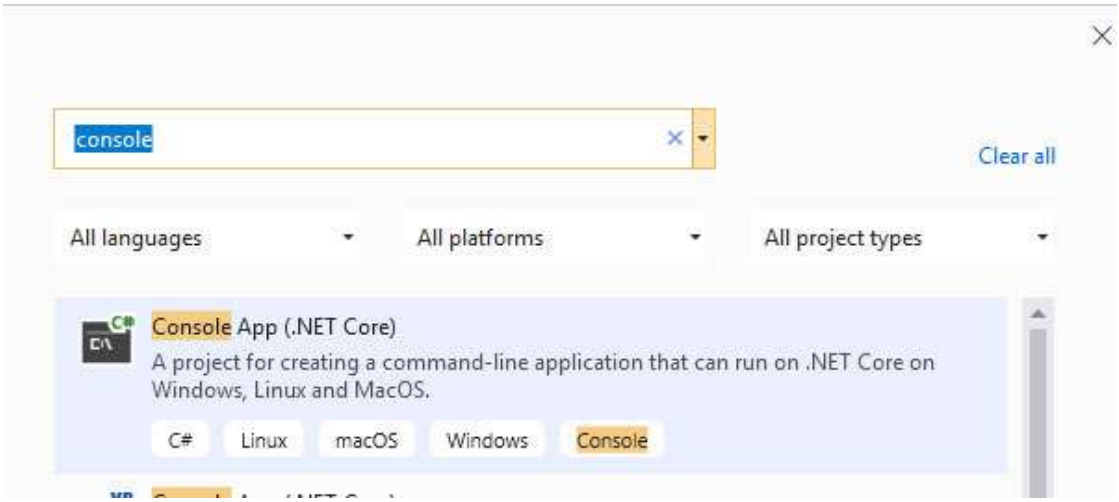


Figure 1.9: *Console Application*

If you are unfamiliar with C# language, it will be your first opportunity to get started with the language, structure, keywords, and statements. Once you confirm the project's name in the application creation dialog and make it to the following steps, the result is a simple Console Application containing only one class called Program.cs. To see the content of the class, just open the file by double clicking on it, and Visual Studio will show the class in the code editor section, as shown in *Figure 1.10*:

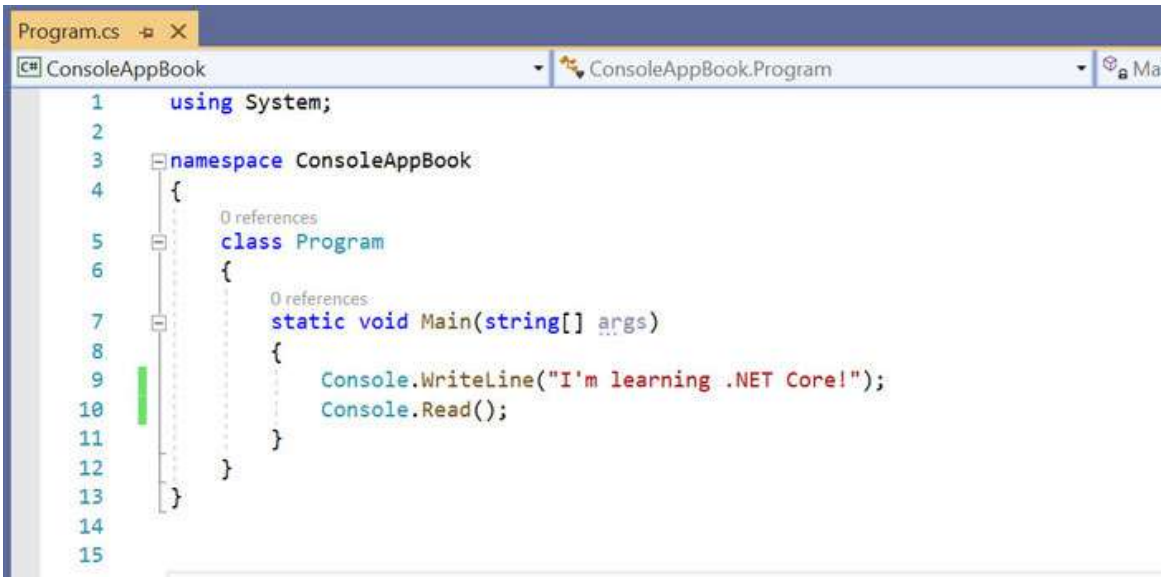


Figure 1.10: *Console Application*

Namespaces organize every C# application, referred to in Figure 1.10 in the first line. A regular enterprise application might have a lot of libraries, packages, and various used namespaces for different purposes. Their names and paths follow a logical sequence hierarchically, and they might belong natively to the C# language as the namespace System. Other ones could be long to a custom or third-party implementation.

Tip: Following and using good name conventions for namespaces, classes, methods, and variables is one of the most important best coding practices, and it is valid for any programming language. Always use meaningful names across the system, applying the Camel Case pattern for classes (for example, MyBpbBook) and Upper Case for constants (for example, MAX_ITEMS_ORDER = 3). Make your code easy to read by others and even by yourself.

Line 3 contains the custom namespace reference for our project (ConsoleAppBook), and it usually is the name of the project or the company organization name, depending on the architecture of the application and the technical decision made by the team. By default, the Console application has a main class called Program and a static method called Main, which will run once the application is started. The content of the Main class in the example is a Console.WriteLine, a method that will print the content “I’m learning .NET Core” on the screen and the Console.Read will ask the user for a typed entry.

Once it is running by Visual Studio using the option “Start Debugging” in the Debug superior menu option, the application starts, and it shows in the console the printed phrase specified in the method “Main,” as shown in *Figure 1.11*:

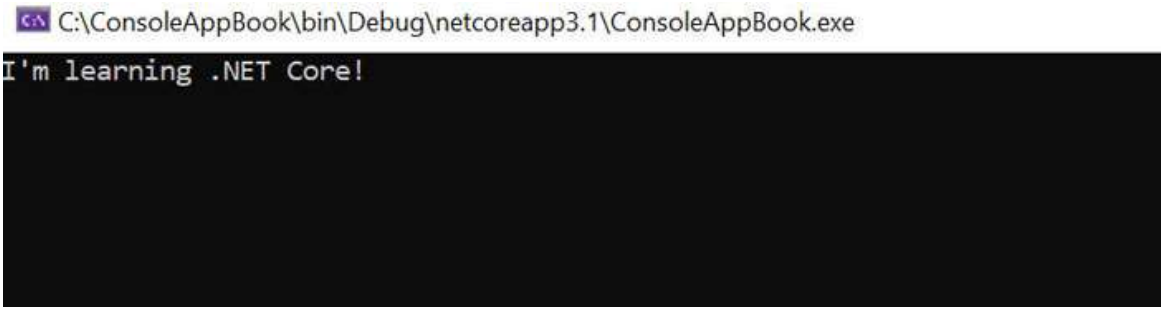


Figure 1.11: *Console Application debug*

Another known type of project template in the .NET platform is the **Windows Form** project, which allows us to create rich desktop applications using visual components, easily integrated with native desktop components for Windows and other platforms. The first version of .NET Core (1.0 and 2.x) did not support Windows Form applications, but it was fully introduced in .NET Core 3.0 version with relevant improvements if we compare it to the legacy .NET Framework version. One of the most relevant improvements is related to self-contained executable files. The application is built with self-contained mode enabled, and the .NET Core will generate a single file with all the libraries and files the application uses. The executable will contain all the necessary project dependencies, such as .NET Core libraries and third-party implementations. Therefore, the machine where the application will be installed does not need to have any extra library or .NET Core installation. The executable contains everything necessary. It facilitates the deployment, installation, maintenance, and even software updates in any environment, and it works appropriately for Windows operating systems and even for Linux and macOS.

To get started with the creation of Windows Form applications, you must go to the creation template dialog already shown in this chapter, search for windows form, and the results will show the Windows Form project type as a suggestion, as demonstrated in *Figure 1.12*:

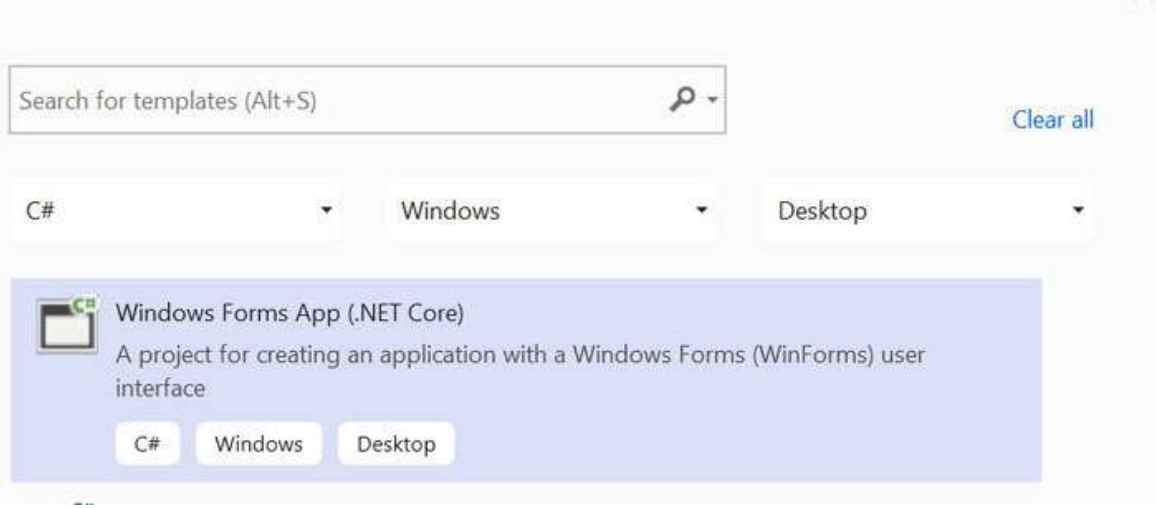


Figure 1.12: *Windows Form*

Visual Studio provides an extensive list of visual components that can be used in Windows Form applications. The available list is helpful because almost all enterprise applications share the same or at least similar visual components. Among them are specific ones for common form controls such as TextBox, Label, and ListView. It also contains many options for native operation systems features such as media player, embedded browser, file and folders manipulations, and much more. To access the complete available list, go to the left sidebar on Visual Studio and open the Toolbox tab.

You can drag and drop components directly into the form and change their properties visually or access the property tab, selecting a component and clicking on the Properties tab in the right sidebar, as shown in *Figure 1.13*:

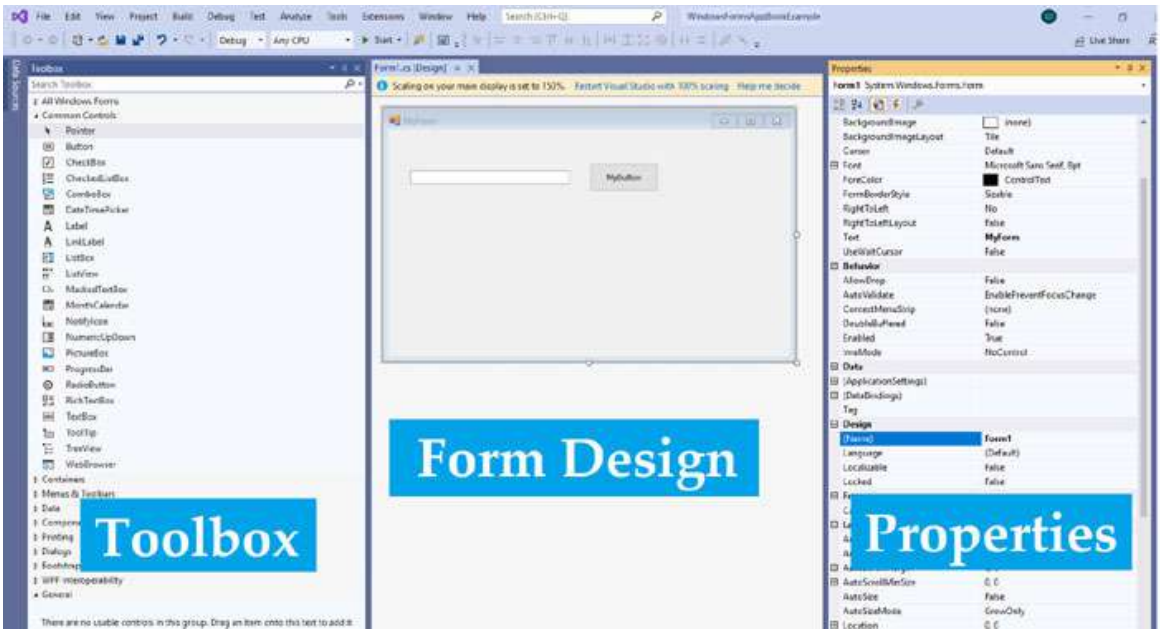


Figure 1.13: *Windows Form*

To include a logical statement to your components, you must double click on the control to access the source code control associated with the component. Each visual control has custom events associated with the component type. Considering you can start your logical coding in a design mode, Windows Forms is one of the easiest ways to learn C# language because you can quickly see the results of your code working. Also, once the project is created in Visual Studio, the program is ready to run without extra configurations, increasing productivity and deployment. As a cross-platform solution, a Windows Forms application built in .NET Core is ready to be used and installed on any platform, such as Linux, macOS, and Windows operation system.

To get started with Windows Form application development, once you have created the application on Visual Studio, drag and drop a TextBox and Button controls to the form, and you can position those as you want. Selecting each control, just go to the Properties tab on the right sidebar and change the "**`Name`**" property to "**`txtName`**" for the TextBox and "**`myButton`**" for the button control. After doing that and pressing the "**`Start`**" button on the superior toolbox, the Windows Form application will run, and the design is going to look like *Figure 1.14*:

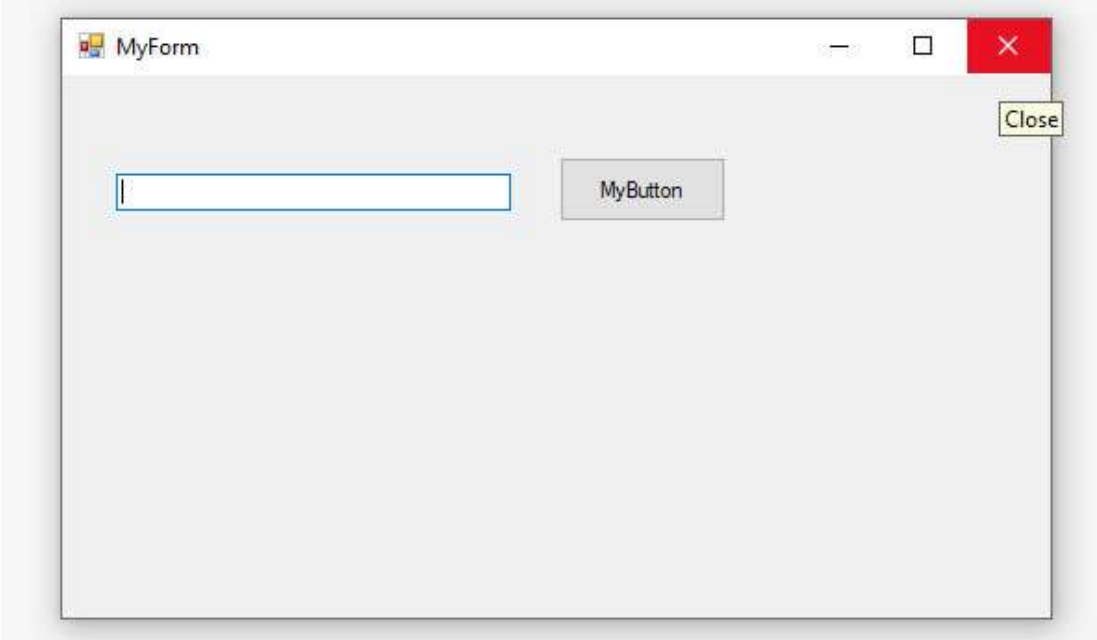


Figure 1.14: *Windows Form*

As a concept and following a programmer's perspective, the Windows Forms project type was made simple and fast to create. Unlike Web development, you can create in visual mode all the screens of your system and code just the logical part of the events, methods, and buttons such as database call operations, communicating with external API's or even making business operations to meet the user system requirement.

Once you have brief contact with two desktop project types in .NET Core (Console and Windows Forms), there are other project types that you must know regarding Web Development. The first is the **Asp.Net Core Web API** project type, which allows us to create **Application Programming Interface** (**API**) applications based on REST, meeting all the most modern requirements for Web Services, security, and performance.

Tip: REST is the contraction of Representational State Transfer, a standard architecture to build and define Web Services. The most significant advantage of this approach is the interoperability across many languages and platforms. Also, modern Web APIs are mainly based on JSON (JavaScript Object Notation), a broadly used pattern for data object transfer between systems. All modern languages provide methods to handle JSON objects properly. It became the most popular way to integrate systems.

To create a Web API project in .NET Core, you must search Asp.Net Core on the search box in the Visual Studio project creation dialog, give it a name, and it will be redirected to the list of available Web project types in .NET. Choose the API type and finish the final step, as shown in *Figure 1.15*.

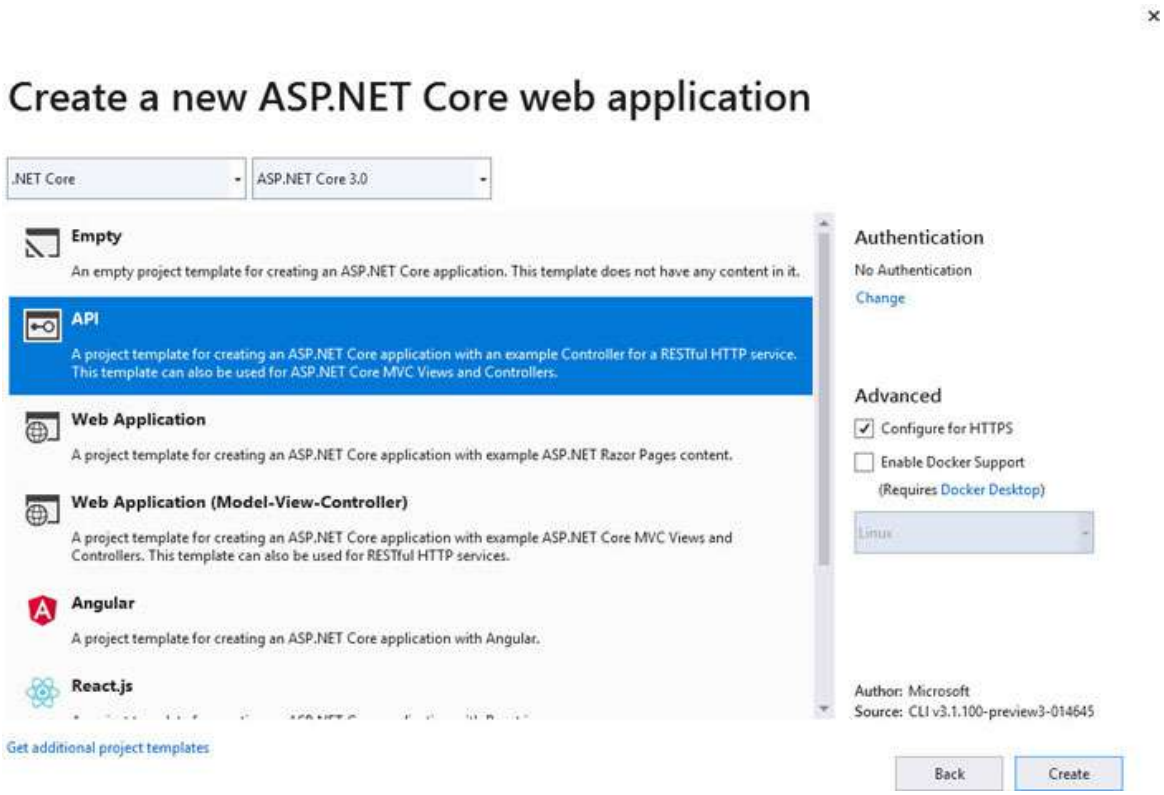


Figure 1.15: *Windows Form*

Once the creation is complete, the Visual Studio generates a Web API with a basic structure based on Model View Controller architecture, which will be explained further in this book. Each API has multiple methods called endpoints as well. Each exposed method could contain security settings, allow HTTP verbs to be configured, and also logical programming to make the necessary operations according to the main target of each endpoint. To create a new endpoint, you must take these steps:

1. Create a new Controller by clicking the right mouse button on the Controller folder shown in the Solution Explorer right sidebar. There are a few Controller types, but it needs to be of the kind of API Controller Empty.
2. Give a name to the Controller, for instance, “CustomerController,” and save it.
3. The Visual Studio will generate a Controller class without a method. To insert one, create a public method that returns a string type and an

underlying string content for the test.

4. After creating a public method, it is necessary to specify the HTTP Verb allowed for this method instance, **`HttpGet`**, which specifies that this method should be called using the GET method by the application which will use this API. Usually, different HTTP Verbs are used for distinct purposes: for example, POST is used for inserting data, and GET retrieves data by the API.
5. Once all the steps are done, your code will look like *Figure 1.16*:

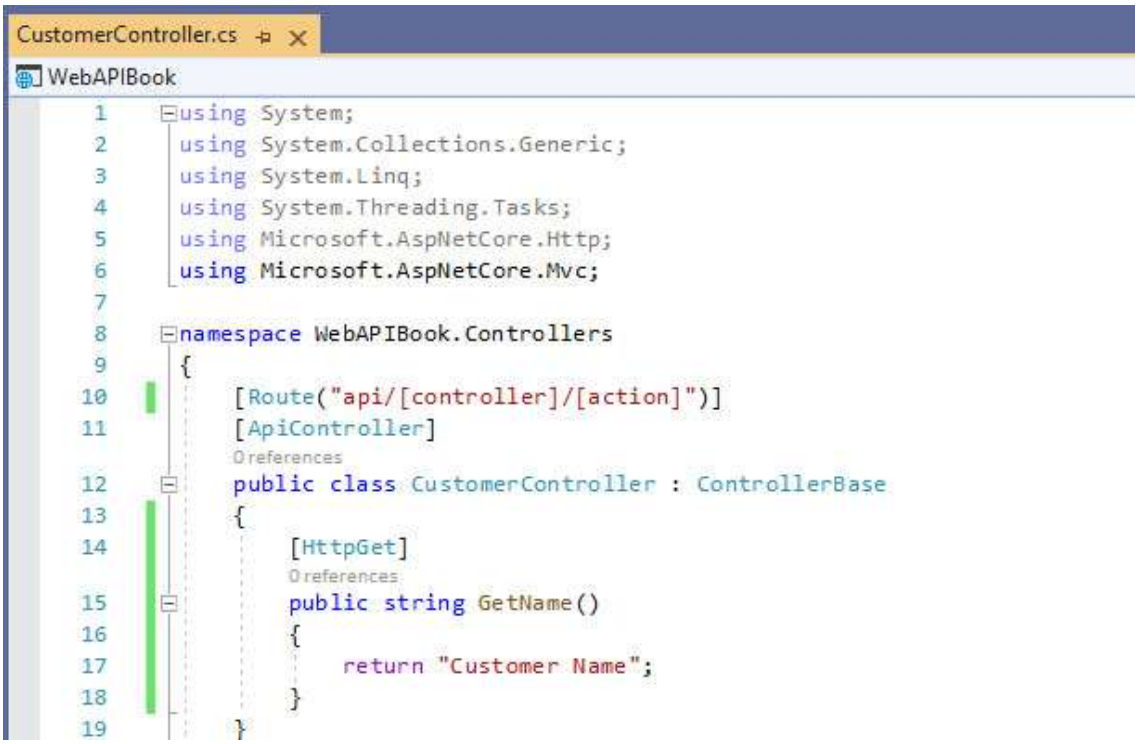


Figure 1.16: *Web API code*

For organizational purposes, all the controllers in a Web API project usually share the same namespace as WebAPIBook., as you can see in line 8. Also, each Controller must use Annotations above the class name to indicate to the .NET Core application "**`ApiController`**" that this Controller should be accessed as a Web API endpoint. Additionally, you can specify the route or path for accessing all the methods in this Controller. In this example, the route is the domain followed by the suffix API, controller name, and method name, as shown in line 10. After running the project, the path for this example will be **http://localhost/api/customer/getname**. Running the application over this URL, the result is the one shown in *Figure 1.17*:

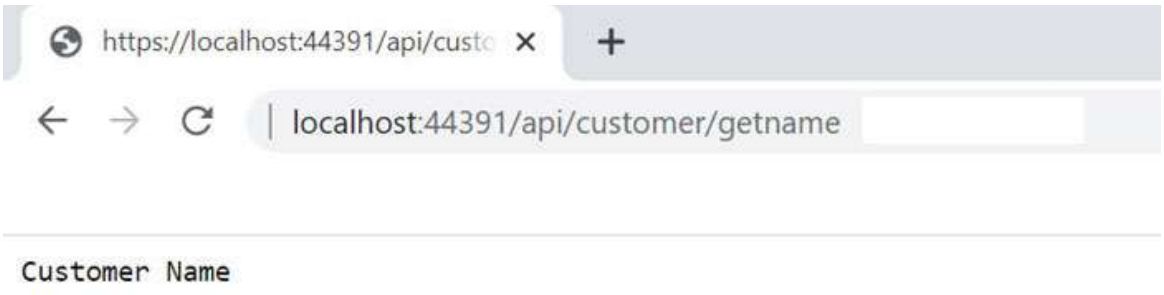


Figure 1.17: *Web API result*

The API method returned the content specified on it, respecting the type of return and the custom logic implemented. In more complex scenarios, which you will implement further following the samples in this book, the API would get a customer name connecting to a database and populate a Customer object to return the data. The Controllers usually contain logical programming targeting other layers in the applications.

Another common project type in .NET Core is the traditional Asp.Net Core **Model View Controller** (**MVC**) project, one of the most popular templates in Visual Studio, considering that it has been primarily used by many companies, even for the migrated projects from legacy .NET Framework versions.

Tip: The Model-View-Controller pattern is one of the most famous architectures for Web applications, not only in the .NET Core platform. This approach follows essential programming practices, such as separating responsibilities between application layers. It is helpful to scale the system in complexity and keep parallel teams working on the same project without huge impact or merging issues.

To get started with the Asp.Net Core MVC project, you must create a new project of that kind on Visual Studio. In the creation project dialog, as shown in *Figure 1.18*, search for Asp.Net and choose the Asp.Net Core Web Application option.

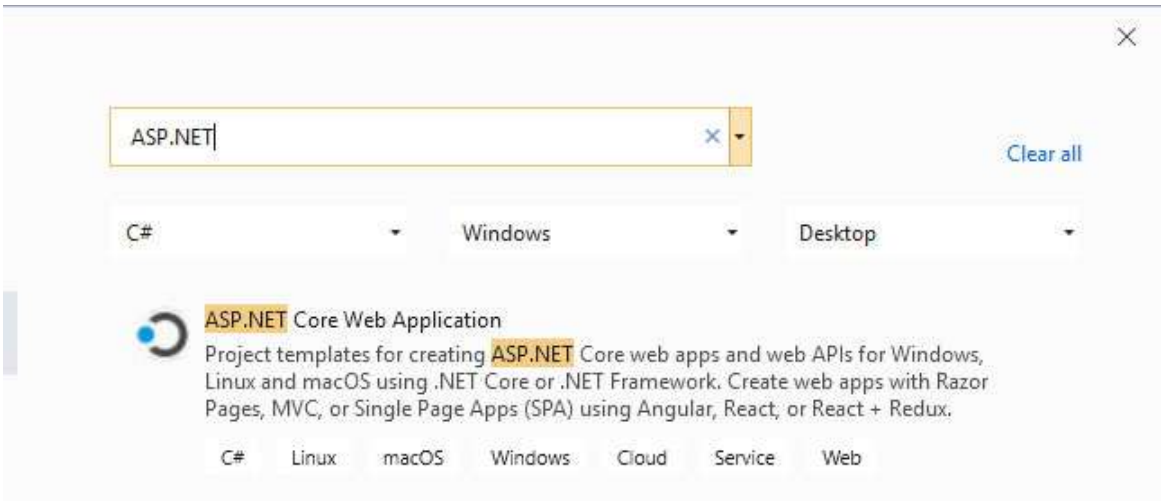


Figure 1.18: *Asp.Net Core MVC project*

In the next step, after defining the project name, choose the template called Web Application (Model View Controller). Once the project is created, Visual Studio generates the default folder and file structure for a standard MVC application. The project has three main folders: Model, View, and Controller, respectively correspond to the MVC architecture.

The **Model** is responsible for keeping the state of the application according to specific system requirements, and the other layers of the application are responsible for populating and managing the model correctly. The **View** is responsible for rendering the user interface, such as HTML, CSS, and JavaScript. By default, in Asp.Net Core MVC projects, the Razor engine is used to mix .NET Core code with HTML. This approach makes it possible to implement and use logical programming in C# inside the Views and interact with HTML elements. Finally, the **Controller** manages the routes and user actions such as page requests, form submissions, and other data request types and user interactions. Once the user requests an action, it is managed by a specific Controller depending on the path specified and associated with it. Each Controller could contain and support many actions, each having its responsibility and logical programming implementation.

To create your first custom MVC application, take the following steps:

1. In Visual Studio Solution Explorer, click on the right mouse button on the Models folder and choose the option to add a class giving the name “Book.” It will generate an empty book class. You must create the properties of the class, as shown in *Figure 1.19*:

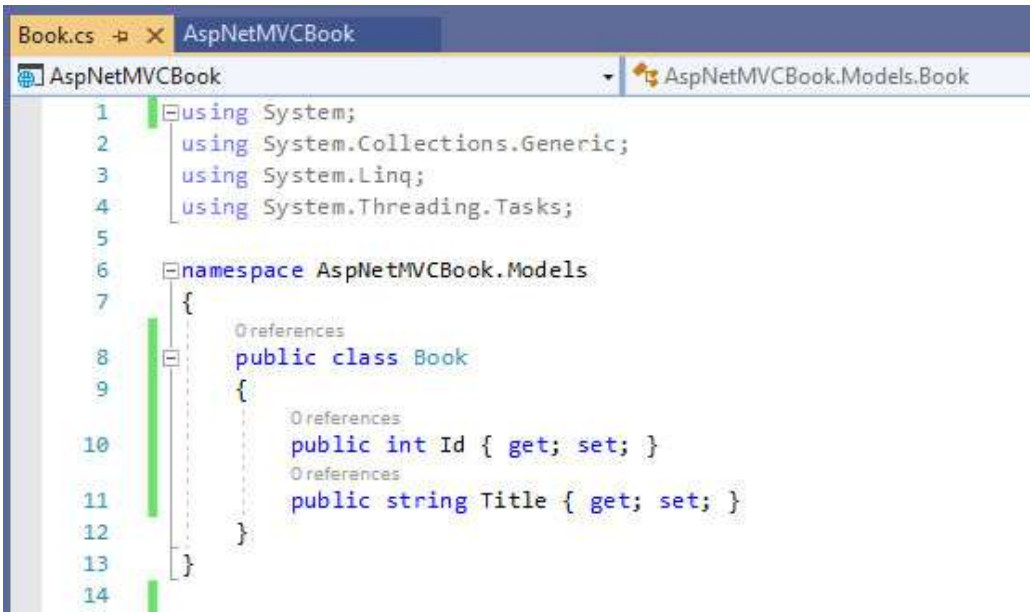


Figure 1.19: *Book Model*

As you can see in *Figure 1.18*, two properties were created for the Book class: Id and Title. They will be used to render the book information on the View after the Controller has done the logical process.

2. On Visual Studio Solution Explorer, click the right mouse button on the Controllers folder and choose the option to add an empty MVC Controller, giving the name "BookController." It will generate an empty controller class. By default, the scaffolding functionality generates an Index method without implementation. As the purpose of this example is to show a book in the View, we must implement it in the Controller, returning to the View an instance of the Book model class. To get this result, the class should have the following implementation in *Figure 1.20*:

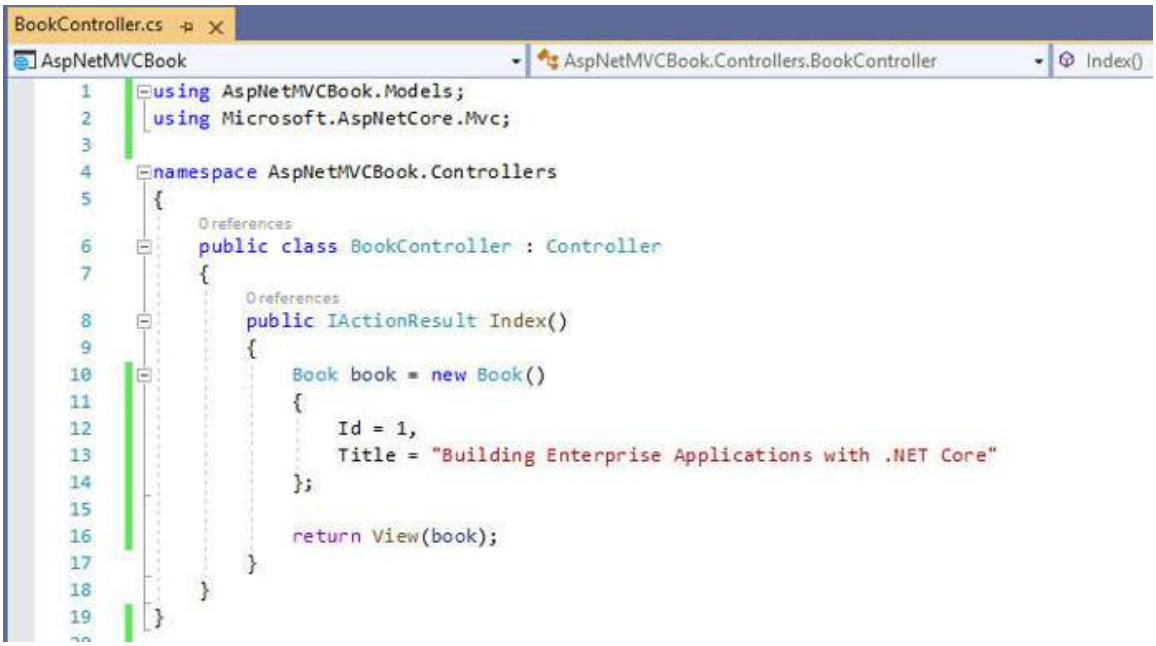


Figure 1.20: *Book Controller*

As you can see in *Figure 1.20*, an instance of the Book class was created inside the Index method with the properties populated with specific sample values. As the View will use that information, the book object was passed as a parameter in the return View instruction in line 16.

3. The last step is creating the View to have a complete MVC sample. Click the right mouse button on the Views folder and create a new folder called "Book." In Asp.Net Core MVC applications, the Views folder is strongly associated with the Controllers. All the Views that the specific Controller will handle should be placed in a folder with the same controller name inside the Views folder. Each action that returns a View must be present in the view folder associated with the Controller. After creating the book folder, add a new View file by clicking the right mouse button on the Book folder and choose the option View giving the name "Index," whose value is the same as the existent method in the Book Controller.

   Considering the View contains HTML code and can interact with the server-side code using the Razor engine, a View must mix Razor code syntax and the necessary code to make the presentation part of our project. Also, as the sample controller has a method that returns a populated book object to the view, you must refer to which type of model is associated with the view at the top of the file, and you can call the model properties using the Razor statement, as shown in *Figure 1.21*:

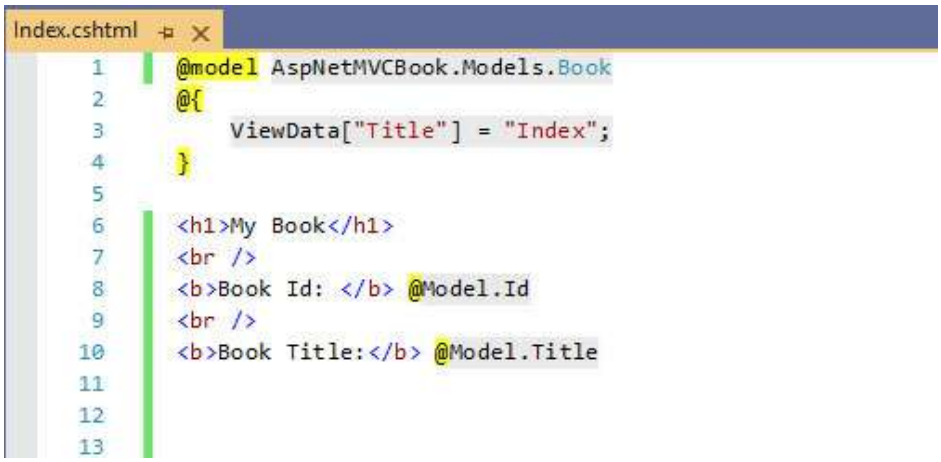


Figure 1.21: *Book view*

Between lines 6 and 10, there is the necessary code to show the book information properties with the values populated by the Controller. The View only receives the data from the Controller, and it is not recommended to have that much logic implementation on it, as the primary responsibility of the view is to render data and present it to the user. Finally, after running the project on Visual Studio and typing the path of our Controller (book following by the index), it is possible to see in *Figure 1.22*:

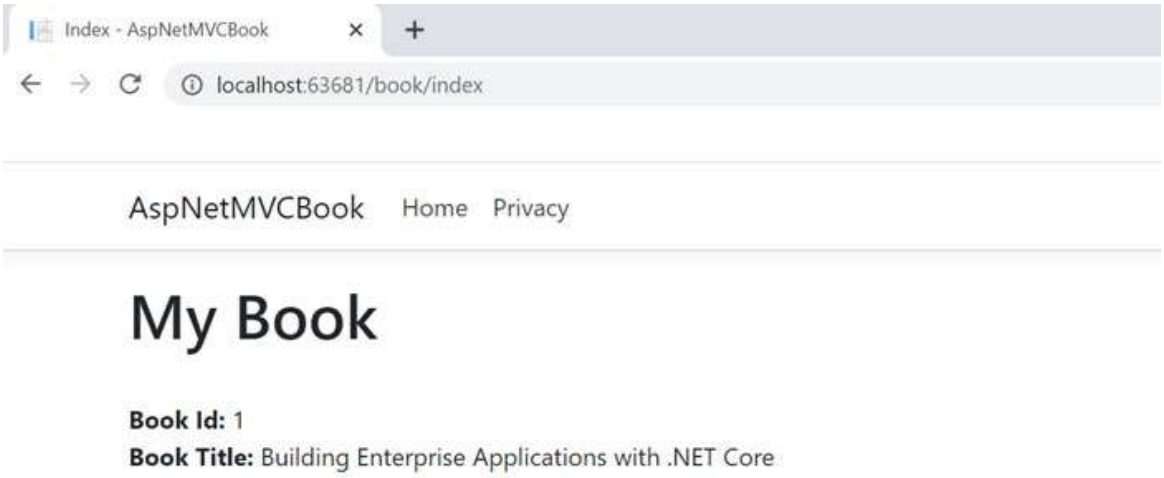


Figure 1.22: *Book view*

Another pretty popular project type for .NET Core applications is the **Razor pages** template, it allows us to benefit from the Razor engine, but with a complete MVC architecture not being mandatory. This type of project is suitable for small and straightforward projects or just needs a presentation layer consuming external

APIs, considering the huge complexity of logical programming is delegated to a third project.

To create a Razor Pages project, go to the Visual Studio creation project dialog, choose the Asp.Net Core Web Application, and then the Web Application type. A simple Web project will be created containing a folder called Pages, where your views will be placed. As shown in *Figure 1.23*, the structure is more straightforward than an MVC project and does not follow any architectural pattern.

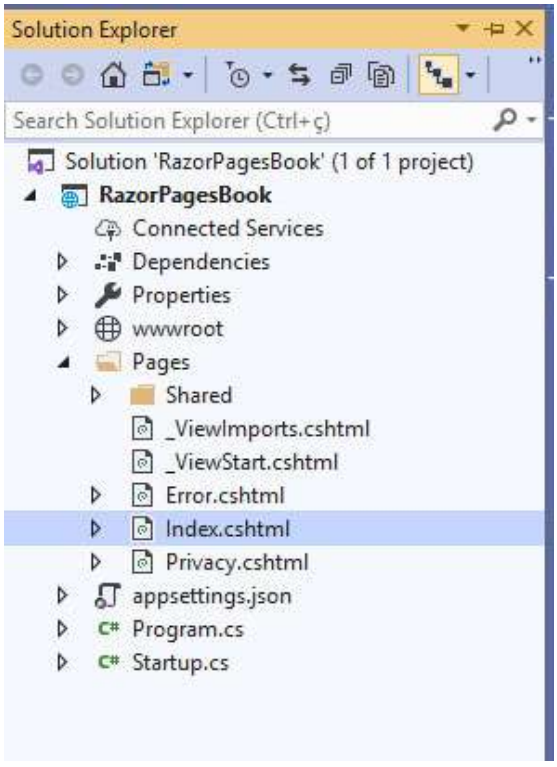


Figure 1.23: *Razor pages*

You must refer to their code inside the Razor Page to use models and controllers associated with the views. Each page contains its logic implementation and can be considered an independent component. It is a good option for simple data input and for non-complex logic operations. Also, this project type increases the time spent in building applications because it does not require configuring routes and strongly naming folders and files in the correct places.

Each view has associated its server-side code file with the same name but with the .cs extension. You must place all the logic implementation and model references in this .cs file, similar to what was done with the Book example. Also, it can have multiple methods associated with the page, different from the MVC approach, where there is a strong relationship between the controller action and the views.

For that reason, the Razor pages project is considered more flexible and manly in simple contexts.

Tip: Considering that many legacy projects were developed using the obsolete Asp.Net Web Forms project type from .NET Framework, it is common to read articles and presentations comparing the old Asp.Net Web Forms and Asp.Net Core Razor Pages. However, they are entirely different in their essence. The Razor Pages are not based on the Postback concept and do not keep the view state in the server in all the page life cycles.

The .NET Core 3.0 officially introduced the **Microsoft Blazor** template to the Web project types. This project uses WebAssembly and Razor engine to create powerful client applications running server-side code directly on the browser. Usually, a regular system based on Web technology renders to the client-side only HTML, CSS, and JavaScript. In a Blazor project, it is possible to run C# code directly on the browser. This type of .NET Core project will be explained in rich detail in *Chapter 14 "Blazor, the Single Page application of /NET Core"*, where a full demo will be made step by step.

Also, among the available project templates on Visual Studio, there are many mobile and game development options. The first one is based on Xamarin, a framework that allows us to create multi-platform mobile applications for Android and iOS using only C# language. It is an impressive engine that made it possible to create a single app and generates the underlying native code for Android and iOS platforms, increasing the productivity and the adoption of the .NET platform by mobile developers who traditionally use Java or Apple technologies to build their Apps. For game development, it is possible to use Unity 3D using C# and reuse all the native libraries from .NET Core.

Conclusion

As seen in this chapter, the .NET platform contains a wide range of project types that allows us to create enterprise applications for multiple platforms and various types of devices, taking all the benefits of the templates already present in Visual Studio installations. Visual Studio is a powerful IDE that contains everything developers need in a single tool, including options to connect to databases, cloud resources, and many more features.

In the next chapter, you will have the opportunity to learn details on the state of the art on the .NET platform in terms of versions, its history, and an overview of current features and possibilities using the latest .NET version.

Points to remember

- Visual Studio Code is a multi-platform, open-source version of Visual Studio, which runs on Linux, macOS, and Windows.
- The Visual Studio IDE is currently available for Windows and macOS operating systems.
- The built-in Visual Studio templates can speed up the development of many project types, such as Windows Forms, Web APIs, Blazor applications, and others.

Multiple-choice questions

1. **Which language given below is not supported in Visual Studio?**
    a. C++
    b. Java
    c. F#
    d. Objective C
2. **Which project type supports the creation and exposure of endpoints over HTTP protocol?**
    a. Blazor
    b. Console
    c. Web API
    d. Razor
3. **Which type of project can be used to share resources and implementation across multiple .NET projects?**
    a. Class library based on .NET Standard
    b. Console application
    c. Windows Forms
    d. None of the above

Answers

1. **d**
2. **c**

3. **a**

Questions

1. Explain the difference between .NET Framework and .NET Core.
2. Describe in your own words the concept of multi-platform development.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

Status of the .NET Platform

Introduction

The .NET platform has transformed into a modern cross-platform technology that allows us to create applications that run on any operating system and multiple devices. Understanding the history of .NET versions is vital to make bold decisions on which version to choose for developing and building world-class enterprise applications.

In this chapter, you will have the opportunity to familiarize yourself with the previous versions of the .NET Framework, find out how the change to .NET occurred, and what the expected future is for this platform.

Structure

In this chapter, we will discuss the following topics:

- History of the .NET Framework
- The .NET Core versions
- From .NET Core to .NET 7

Objectives

After studying this unit, you should be able to understand the current status of the .NET platform. We will also discuss the difference between .NET Framework and .NET Core and understand multiplatform concepts.

History of the .NET platform

The .NET platform was created long before Web applications became popular and even before the first version of the C# language. If we pay attention to the massive adoption of .NET nowadays, it gives the impression that the scenario was always like this, especially for those who did not

experience software development around the beginning of the 1990s. Looking back almost three decades ago, when other programming languages like Java and PHP stood out, the first for generalist cross-platform software development and the second for Web applications, it seemed essential for Microsoft to enter this great market of development tools. The first version of the .NET platform was launched as a beta product in 2000. And finally, in February 2002, the official 1.0 version was released, supporting Windows 98, Millennium, and XP. The most important feature of the first version was the **Common Language Runtime** (**CLR**), which allowed us to create .NET applications using more than one programming language. The .NET platform could execute it using CLR and convert it into **Common Intermediate Language** (**CIL**), also called **Intermediate Language** (**IL**). Therefore, sharing libraries in C# and components in Visual Basic was possible in the same project. Being able to use multiple programming languages in the same program did not necessarily mean being able to create multiplatform software using .NET, whereas it was possible to run and develop .NET Framework applications only on the Windows operating system.

In the second quarter of 2003, the first relevant update to the version .NET framework version (**.NET 1.1**) was released, and the support for **Open Database Connectivity** (**ODBC**) was introduced, following standard requirements for database integrations in .NET applications. Up to this point, the .NET platform did not have wide adoption on the market, and essential features were released to meet the most modern software development demands and trends.

In January 2006, Microsoft launched version 2.0 of the.NET Framework and other vital tools and products, such as Visual Studio 2005, a more stable release of the new SQL Server database 2005, and Biz Talk. One of the most significant improvements of this version was the support for 64-bit computer architecture and new features in C# language, such as a partial class, new authentication options for Asp.Net, and ‘Data table’ objects, which allowed us to make database operations in memory using datasets.

Alongside these new features, Visual Studio was becoming mature and complete, being a fully **Integrated Development Environment** (**IDE**), even for programmers who work with languages different from C# and Visual Basic. Also, the Asp.Net Web Forms started having extensive adoption among developers who had experience with desktop applications and were

migrating to Web development. Visual Studio 2005 already contained features to quickly drag and drop visual Web components and add events to those interactively in the environment. This way of developing applications was very similar to desktop application development. For that reason, developers unfamiliar with Web development could start developing their first Web applications.

The Asp.Net Web Forms occupied an important space in the .NET Framework before the Asp.Net MVC (Model View Controller) was introduced to the platform. Web pages navigation based on forms was common at the beginning of the Web, and it essentially consisted of a user sending a request from the Web browser and the server returning a response, keeping the state of the web controls for giving users a continued experience navigating on the Web page and interacting with its controllers. In that case, the server was responsible for processing the HTML and all the dynamic data. This technology used the **View State** approach to store the values of all server-side components. Additionally, the Asp.Net Web Forms provided the option to use separate files between the HTML and the logic code for each page. That was considered an innovation, and years after that, the separation of responsibilities in the application would be better provided with Asp.Net MVC (Model View Controller).

In November of 2006, the .NET Framework 3.0 was released, including relevant features regarding desktop development and Web Services. At this time, the JSON (JavaScript Object Notation) was not used for integrating services based on Web protocol. The XML (Extensible Markup Language) was primarily used for standard communication between systems, and the **Windows Communication Foundation (WCF)** became the official Web Service of the .NET Framework. The WCF became obsolete once the .NET replaced it with Web API (Application Programming Interface), following the most modern patterns being adopted in the market.

One of the benefits of Windows Communication Foundation was its integration with Visual Studio tools, which allowed us to import third-party endpoints via address and generate classes in C# language correspondent to the objects specified in an external Web Service, even if it was written in another language. The functionality was focused on interpreting the XML of the third-party Web Service specification and creating objects and their properties without being necessary to write any code line. Additionally, the Service Import tool automatically generated the methods in C# language to

open connection and send requests to the external Web Service, parsing the response to objects in C#. It was an extremely productive tool and hugely important for the popularity of the .NET Framework in the market. These days, many companies still have Web Services developed using Windows Communication Foundation as a legacy project. The same situation happens with Asp.Net Web Forms applications; considering many companies adopted it to large projects and did not migrate them yet to a newer version of .NET Framework project types or .NET Core.

Another important technology introduced to .NET Framework 3.0 was the **Windows Presentation Foundation (WPF)** project type, a new way to create desktop applications using XAML (Extensible Application Markup Language) and develop rich user interfaces using 3D computer graphics hardware. It was an alternative to the traditional Windows Form Desktop applications existent in the .NET platform. Despite other project types of .NET 3.0 becoming only legacy projects and not having continued updates until .NET Core, the WPF was still largely used by companies and .NET developers because it continuously received updates and had full support in .NET Core, not only in .NET Framework.

The **.NET Framework 3.5** was released in November of 2007, bringing performance improvements for desktop applications. Additionally, the first version of **Entity Framework** was introduced, enabling better communication with databases and is an alternative to ORMs (Object-Relational Mapping) of other distinct platforms. However, a new Entity Framework release was launched in 2008 at the same time as the .NET Framework Service Pack 1, including updates after technical community feedback, with this version not mature enough to be broadly adopted in many companies across the globe. The ORM concept and its importance were widely accepted on the market, and considerable improvements in Entity Framework became extremely necessary at this point, mainly to give reliability and stability to the product.

After only one year, Microsoft provided version **4.0** of the .NET Framework, but it did not represent any break changes and did not have many new features, different from the .NET Framework 4.5, which was released in August of 2012 and contains many improvements for Web applications, Desktop development and native components for Windows 8 operating system. Among all the new features, the new functionalities for Asp.Net applications stood out, including support for HTML5 for Web Forms and

support for WebSocket protocol, which represented a significant step to follow the most modern Web standard practices at that point. Also, with this version of the .NET Framework was possible to use asynchronous HTTP requests and responses.

Finally, in July 2015, the **.NET Framework 4.6** was released, being the last big version of this platform before the first version of .NET Core, despite having minor updates after that in versions 4.7 and 4.8. The 4.6 version introduced support for Azure SQL Server distributed transactions, improvements related to the Windows Presentation Foundation user interface, and support for the localization of data annotation attributes in Asp.Net.

The .NET Core versions

In 2015, the information technology and software development market was changing to become compatible with cross-platform concepts. The fact that the .NET Framework had support only for Windows operation systems represented a significant limitation for its evolution. Companies largely accepted it, and developers and technical communities that were mandatory for any software development technology to be capable of being executed in any operation system, multiple devices, and interoperability with other technologies and platforms.

Meeting these new requirements, Microsoft launched in June of 2006 the first version of .NET Core, a disruptive platform that allows the development and running of .NET applications on Linux and macOS. The .NET libraries for C# and Vb.NET were rewritten entirely from scratch, including a new version of Asp.Net projects and Entity Framework. They were renamed to Asp.Net Core and Entity Framework Core, following the new brand of the .NET platform.

Additionally, all the .NET Core libraries and their projects became open-source, with the code being shared on GitHub, making it possible for anyone to contribute to the evolution of the technology reporting issues of this new version. This represents a significant milestone for .NET developers, considering only the Asp.Net MVC project was considered an open source. It significantly contributes to .NET adoption by developers from other technologies.

The .NET Core 1.1 was released in November of the same year; it contained many solutions for issues reported by developers and companies and focused on supporting more Linux distributions. Also, the Entity Framework was improved to support Azure and SQL Server 2016 databases.

In August of 2017, the .NET Core 2.0 was launched with the possibility to integrate with legacy .NET Framework libraries using .NET Standard and improve cross-platform support with new features. Additionally, this version brought the new Razor Pages Web project, an alternative to the traditional Asp.Net MVC project type for creating Web applications within the .NET Core platform. At this point, the .NET Core became a mature platform adopted mainly by companies and .NET developers. The support for cross-platform development was the key to .NET Core, joined with the Visual Studio Code and new command line options. Definitely and officially, developers were able to create .NET applications that run everywhere, participating in the evolution of the subsequent versions of it on the GitHub repository and taking the benefits of performance that were added to .NET Core in comparison to the .NET Framework.

The .NET Standard was a common API specification for .NET libraries, allowing us to use it across all .NET projects. It means that different types of projects would be able to share libraries between them with .NET Standard, including implementations between .NET Framework and .NET Core projects. The .NET Core and .NET Standard are different things. The first one is incompatible with original .NET Framework libraries, and the second was made to be a bridge between all .NET versions. Therefore, considering there is no compatibility between .NET Core and .NET Framework, the best strategy to migrate .NET Framework applications to .NET Core is to create libraries in .NET Standard. In that way, it is possible to keep your old implementation without breaking changes and use the same libraries in .NET Core projects in case migration gradually occurs.

As you can see in the following image, the .NET Standard represents the intersection between .NET Framework and .NET Core projects, and it already contains relevant compatibility in version 2.0, as shown in *Figure 2.1*:

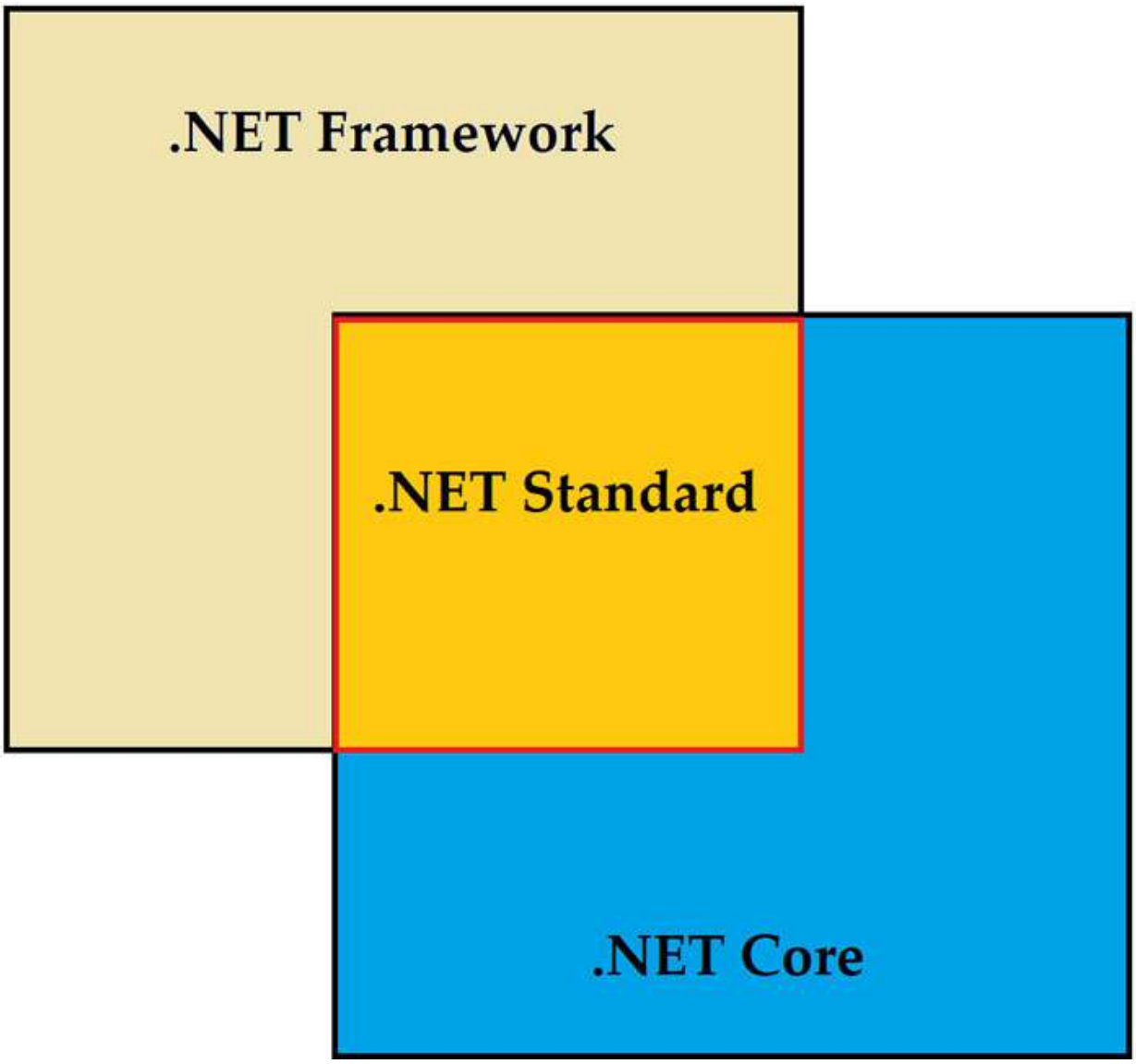


Figure 2.1: *.NET Standard*

Each new version of .NET Standard added universal compatibility in .NET projects in general: .NET Core, .NET Framework, Mono, Xamarin, and Unit. Its specification is the base feature for any .NET library considering all the products, including mobile development and games. The .NET Standard 2.0 supports interoperability with .NET Framework 4.6.1 or later versions. Therefore, projects that use an older version of the .NET Framework, such as 2.0 or 3.5, cannot use .NET Standard libraries. It is recommended to migrate the project to the 4.6.1 version before starting any migration to .NET Core.

To understand correctly how the .NET Standard works and consider how important it represents in migration to the .NET Core of old legacy systems, we must create a project to practice and demonstrate it. First of all, create a project of Asp.Net Core Web Application type on Visual Studio 2019 and choose the option as shown in *Figure 2.2*:

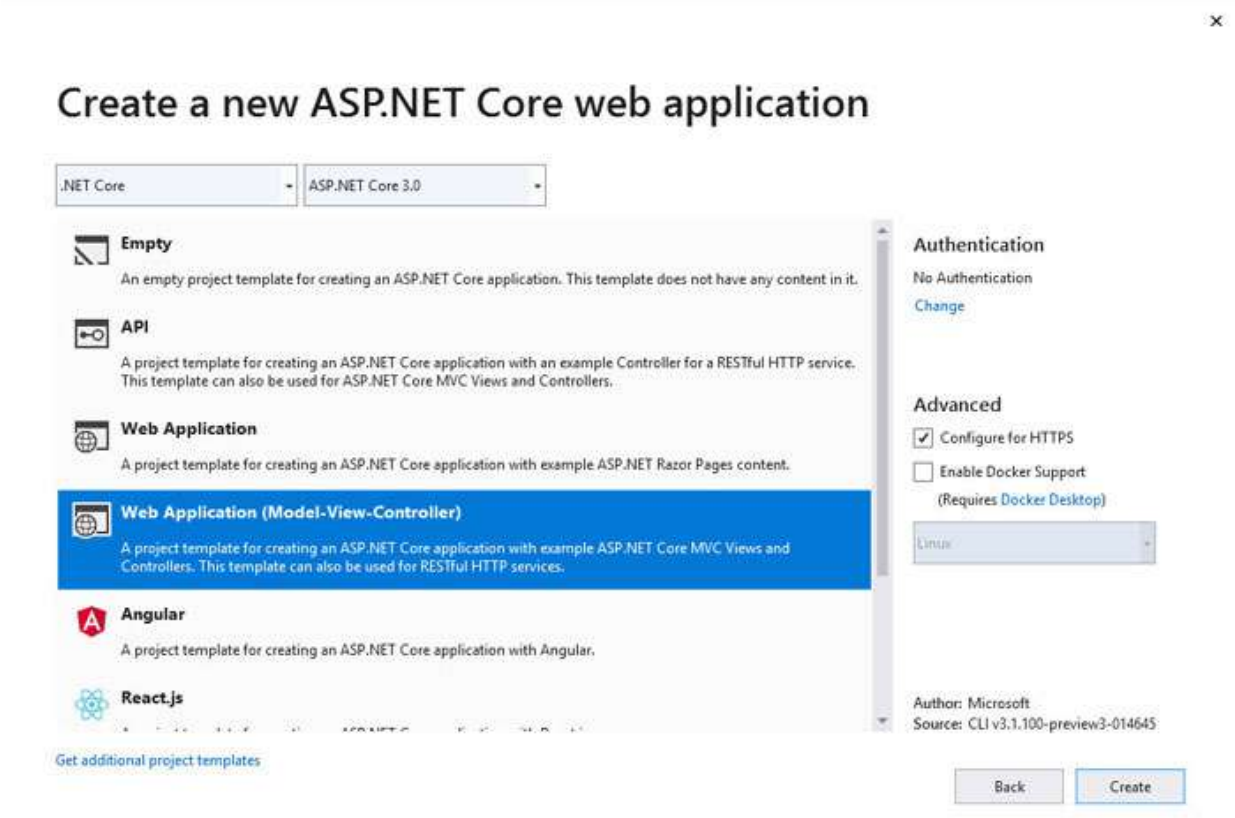


Figure 2.2: *Asp.Net Core Web Application*

After that, create in the same solution a Console App project from the .NET Framework version, even if they are incompatible with each other because of the .NET version. To create the project, go to the project creation dialog and choose the option Console App (.NET Framework) as shown in *Figure 2.3*:

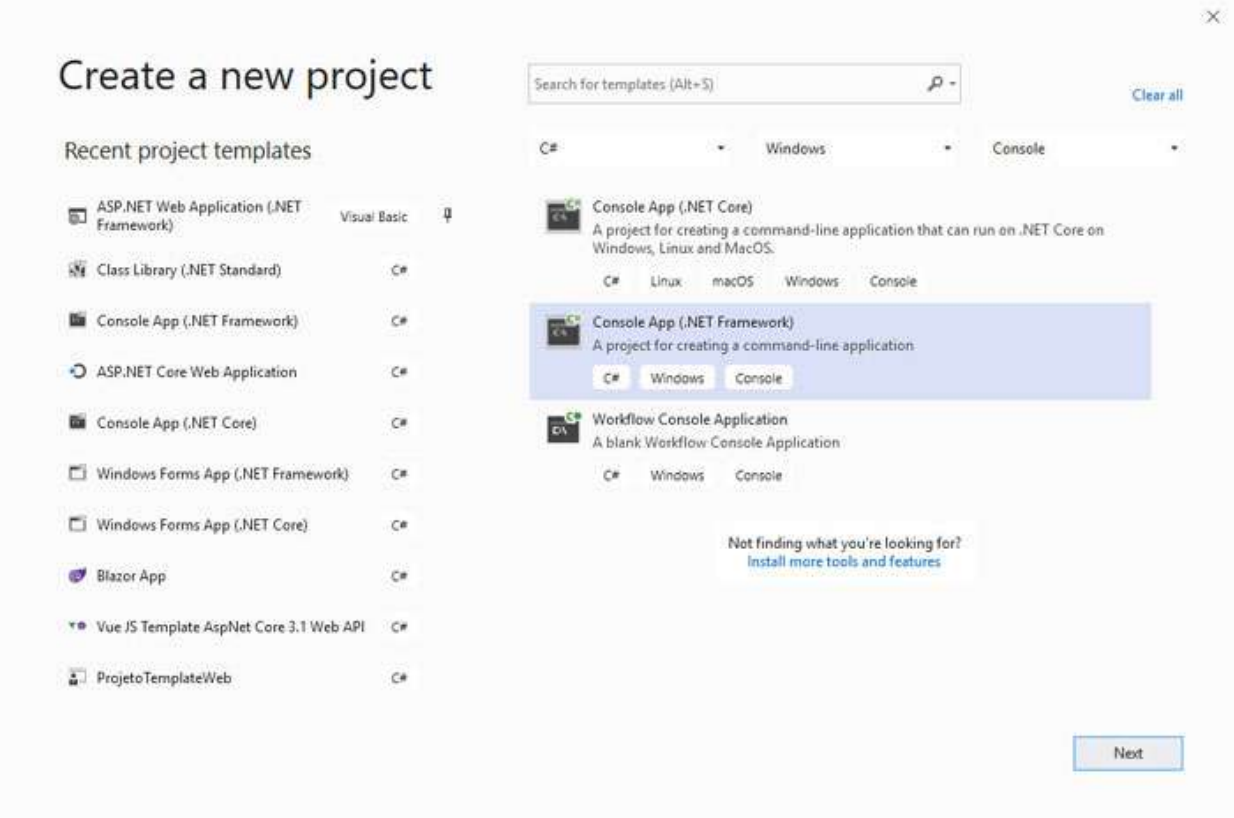


Figure 2.3: *.NET Framework Console Application*

Lastly, create a .NET Standard library in the same solution but in a separate project. In the project creation dialog, search by ".NET Standard" and choose the correspondent option, as shown in *Figure 2.4*:

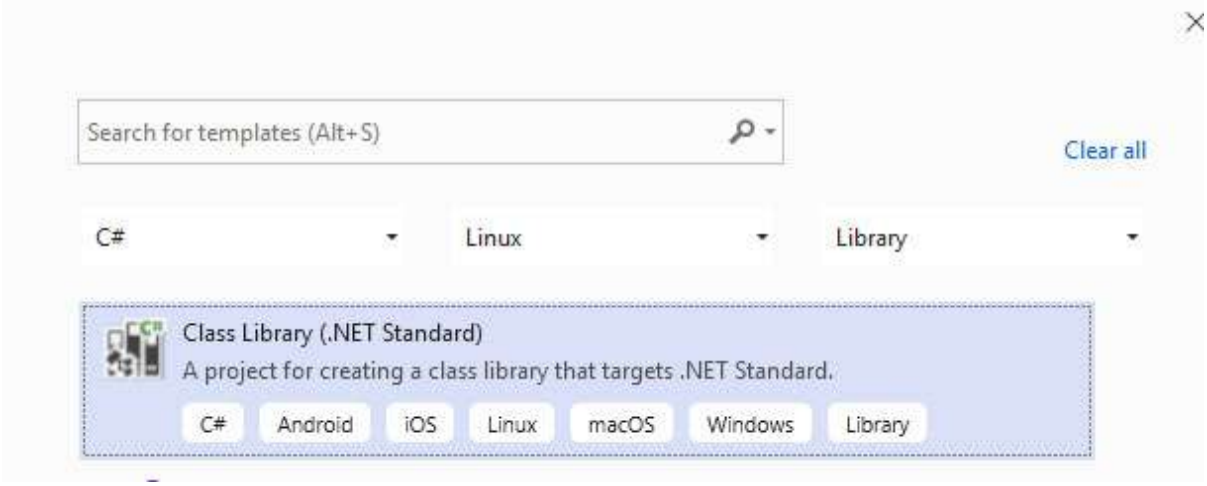


Figure 2.4: *.NET Framework Console Application*

Once these three projects are created, you have the following projects in your solution:

- Asp.Net Web Application in .NET Core version
- Console Application in .NET Framework version
- Library in .NET Standard

The purpose of this example is to show how we can use routines written in .NET Standard and share it between .NET Framework and .NET Core projects simultaneously, demonstrating its compatibility. To test that, you must add the reference to the .NET Standard project library in the other two projects. To do that, double click on the right mouse button on each project and choose the option “Add Reference.” With the reference dialog opened, check the .NET Standard project, as shown in *Figure 2.5*:

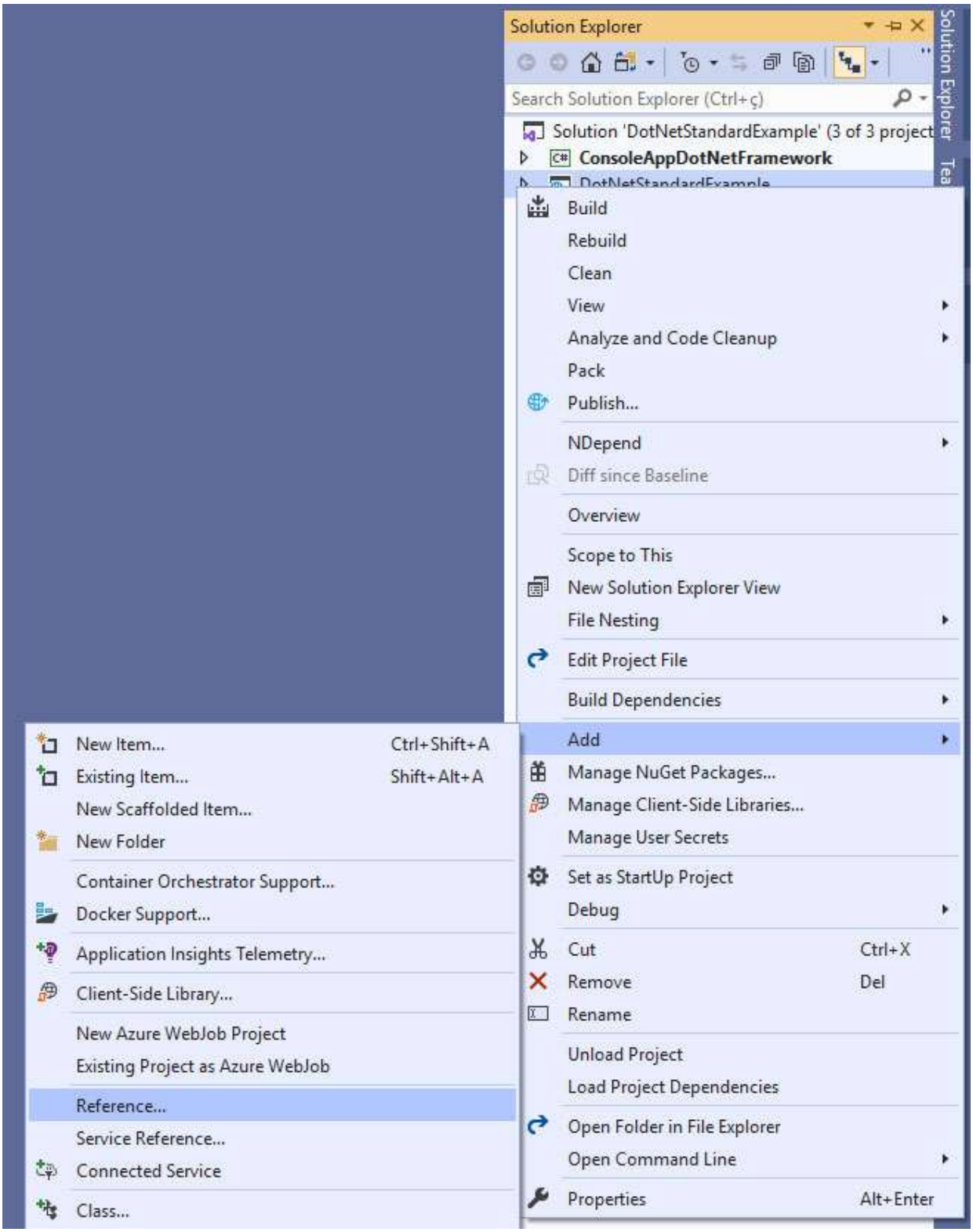


Figure 2.5: *Project references*

The .NET Standard library project needs to be checked in the project references screen, as shown in *Figure 2.6*:

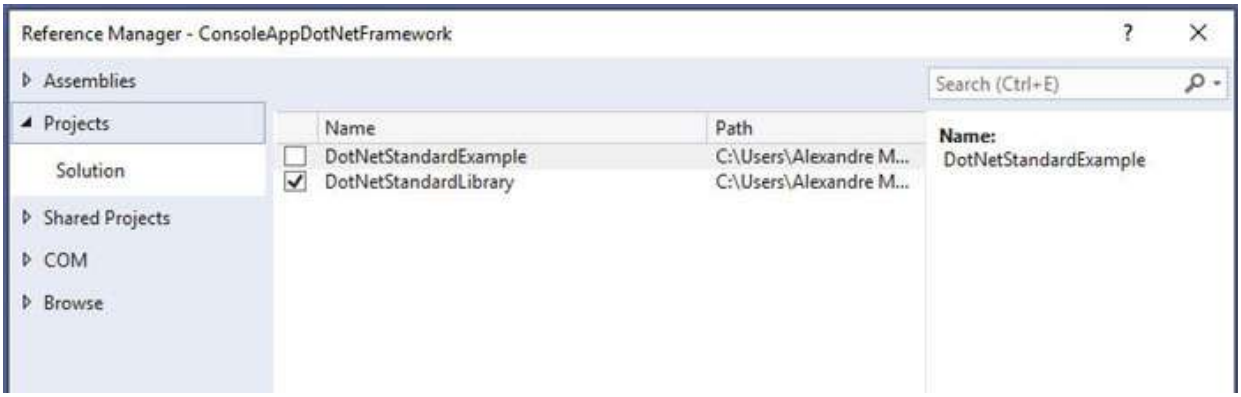


Figure 2.6: *.NET Standard project reference*

After applying these steps for the Console application and Asp.Net Web project, create a class in the .NET Standard library project with the following implementation, as shown in *Figure 2.7*:

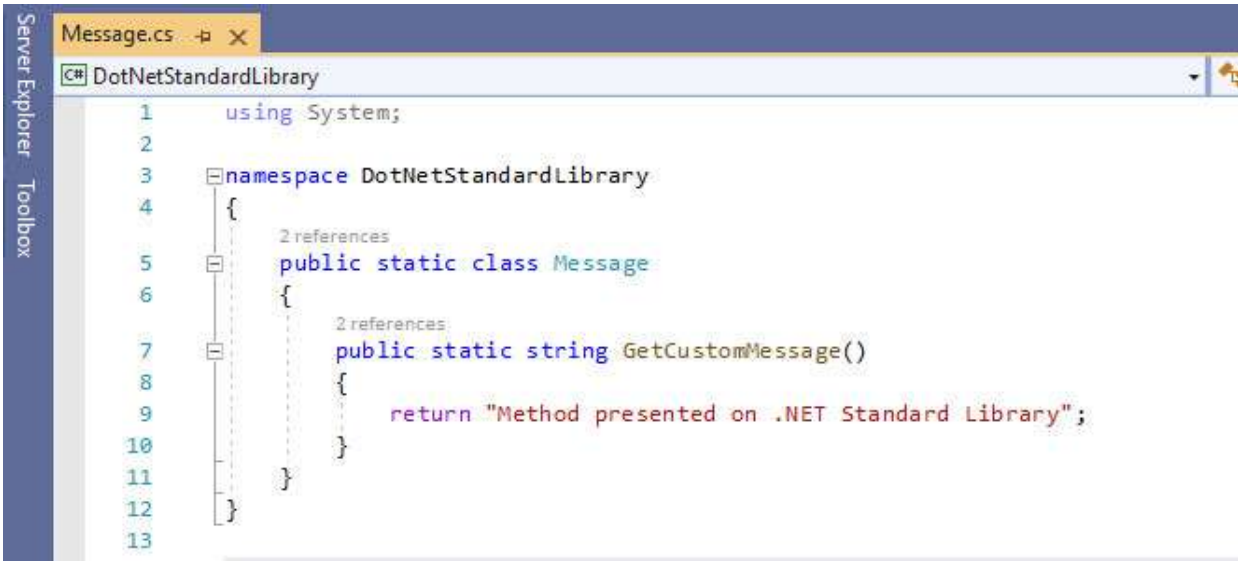


Figure 2.7: *Message class*

As seen in the previous image, there is a C# static class called "**`Message,`**" containing a single method called "**`GetCustomMessage`**" that returns the message "Method presented on .NET Standard Library." The next step is to use that class and its method in both the other two projects. Go to the Web Project and in the Controllers folder, open the file **`HomeController.cs`** and write the code inside the Index method, as shown in *Figure 2.8*:

```
21      public IActionResult Index()
22      {
23          ViewBag.Message = DotNetStandardLibrary.Message.GetCustomMessage();
24          return View();
25      }
```

Figure 2.8: *Message class in the Web Project*

Further, go to the Views/Home folder and the **`Index.cshtml`** file and replace the existing code with the following one, as shown in *Figure 2.9*:

```
Index.cshtml
1   @{
2       ViewData["Title"] = "Home Page";
3   }
4
5   <div class="text-center">
6       <h1 class="display-4">@ViewBag.Message</h1>
7
8   </div>
9
```

Figure 2.9: *Index file*

After applying these changes and running the project, you will be able to see the message coming from the .NET Standard project on the page, as shown in *Figure 2.10*:

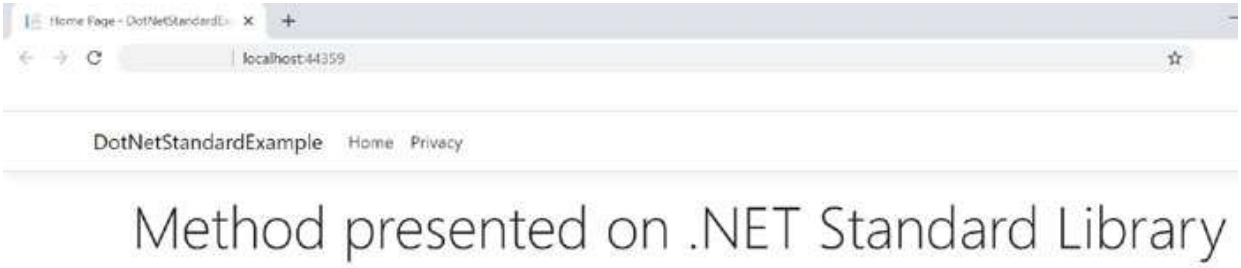


Figure 2.10: *Message displayed on Web Project*

Considering that the compatibility between .NET Core and .NET Standard was verified, the next step to check its compatibility is to use the same method from the .NET Standard on the Console application created using the .NET Framework version. On the Console Application project, open the **`Program.cs`** file and replace its content with the following code, as shown in *Figure 2.11*:

Figure 2.11: *Message displayed on Console App project*

As done in the Web application, the Console application uses the method "**`GetCustomMessage`**" from the .NET Standard project. After running the project, the message is displayed as shown in *Figure 2.12*:

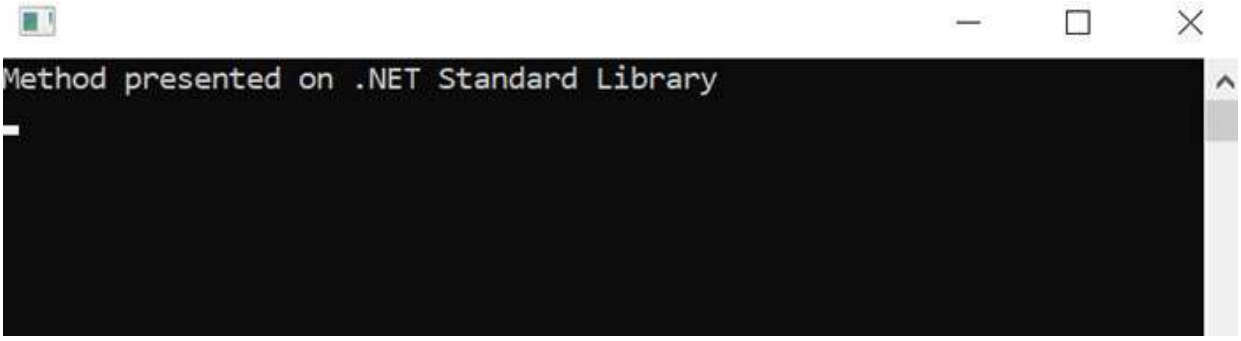


Figure 2.12: *Message displayed on Console App project*

It is essential to realize that both projects (Web and Console) used the same implementation from the .NET Standard. Therefore, that is the best way to gradually migrate .NET Framework projects to .NET Core, being possible to share libraries between them using .NET Standard.

From .NET Core to .NET 7

Since the first version, the .NET Core had many improvements and significant new features were released at the same time as the compatibility with cross-platform development was strictly followed. The .NET Core 3.0 represented the most significant jump in the creation of modern software,

considering in this version were added support to Windows Form application, Windows Presentation Foundation, Artificial Intelligence, and development for **Internet of Things** (**IoT**). In the following chapters of this book, you will have the opportunity to dive deeply into all these types of projects and understand the performance improvements that were applied to C# 8.0, which is part of this .NET Core version.

Microsoft announced that from the second semester of 2020, the .NET platform would be renamed in the next version, being called.NET 5 instead of .NET Core 4.0. It means that all .NET libraries are fully migrated to cross-platform development at this point, and the intention to have one hundred percent of .NET projects as open-source ones will be met.

In November 2020, Microsoft released the .NET 5, which represented the beginning of a new era in software development using the .NET platform once all the efforts for cross-platform development reached a good level of maturity and stability. This means that the most relevant .NET libraries were fully migrated to have cross-platform compatibility at this point.

Among all .NET versions, .NET 5 represents an evolution of .NET Core 3.1 and .NET Framework 4.8, a massive milestone in consolidating all the efforts regarding cross-platform development that started in the .NET platform since .NET Core 1.0. Therefore, it is possible to say that .NET 5 is a unification of the .NET ecosystem for modern and cross-platform development, combining new features and relevant performance improvements.

The unified platform has the cross-platform focus as a key point, which does not only mean compatibility across multiple operating systems, but in terms of multiple devices and platforms, such as Desktop, Web, Cloud, Gaming, IoT, and others. The achievement of multiplatform development using .NET required a complex re-architecture of the entire .NET platform, bringing at the same time significant improvements in terms of performance for existing features for C# language, including type conversions, parallelization, and much more.

In November 2021, the .NET 6 was launched by Microsoft, which included further performance improvements, Arm64 support, .NET MAUI, C# 10, new project templates, support for HTTP/3, improvements on code analysis, DateTime and time zone improvements, and much more. At this point, many

companies were already using .NET 5, and the migration to .NET 5 did not represent a massive challenge for legacy projects.

Finally, in November of 2022, .NET 7 brought significant improvements for modern client development with .NET MAUI, Cloud Native improvements, and new tools to upgrade existing .NET applications.

Conclusion

The .NET platform has incredibly changed over the last four years with the idea behind .NET Core. In the meantime, many developers could contribute to its evolution, and innovations were added on each release, making it a powerful platform to build enterprise applications capable of running in any operation system, various distinct devices, and at the same time, bringing high-performance to the applications and following the most modern market trends for software development.

In this chapter, you understood the current status of the .NET Platform and the main features introduced in each version. NET. Additionally, you learned the best strategy to migrate .NET Framework projects to .NET Core, creating .NET Standard libraries.

In the next chapter, you will have the opportunity to build cross-platform applications using .NET Core and get familiar with the deployment in Linux and macOS of the main project types, such as Web and Desktop applications.

Points to remember

- The .NET Standard can be used to create shared libraries between .NET Framework and .NET Core.
- The .NET Core became an open-source and cross-platform technology.
- The next version of .NET Core after the release of 3.0 will be called .NET 5.

Multiple choice questions

1. **Which version of .NET is not cross-platform?**

    a. .NET Standard

b. .NET Framework 4.6
c. .NET Core 3.0
d. .NET Core 1.0

2. **In which version of .NET Core the Windows Form support was released?**

    a. .NET Core 2.0
    b. .NET Core 2.2
    c. .NET Core 1.0
    d. .NET Core 3.0

3. **What type of .NET project can be used to create libraries compatible with .NET Framework and .NET Core?**

    a. Console Application
    b. Windows Form
    c. .NET Standard libraries
    d. Asp.Net Core Web Application

Answers

1. **b**
2. **d**
3. **c**

Questions

1. What are the main differences between .NET Framework and .NET Core?
2. Explain the best strategy to migrate projects from .NET Framework to .NET Core.
3. In which version of .NET Core the support for Windows Presentation Foundation was added?

Key terms

- **Cross-platform applications:** Applications that can be developed and executed in any operating system: Linux, macOS, Windows, and others.
- **.NET Standard:** The typical API specification for .NET libraries that allowed shared implementations across multiple and disruptive versions of the .NET platform.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Cross-platform Applications

Introduction

Developing multi-platform applications that run in any operating system and multiple devices represent mandatory requirements for modern enterprise solutions. Historically, the .NET platform was based only on Windows. It was a growing limitation in the scenario where the companies were migrating their applications to cloud architecture and hosting them in servers provided by multiple players on the market. Since the first version of .NET Core, the .NET platform has become a cross-platform technology by following the most modern software development standards and allowing the building of applications for Linux and macOS. Considering how important it is to learn to build cross-platform applications, in this chapter, you will learn how to create web and desktop applications for Linux and macOS using .NET Core and C#.

Structure

In this chapter, we will discuss the following topics:

- Asp.Net Core applications on Linux
- Self-contained EXEs

Objectives

After studying this unit, you should be able to understand cross-platform concepts, discuss aspects of web development for Linux using .NET Core, and create basic desktop applications for Linux using .NET.

Asp.Net Core applications for Linux

Building .NET Core applications for Linux do not require developing the applications in the underlying operating system. It can be done even by

developing on the Windows platform, considering the .NET Core platform is cross-platform. Therefore, we can code our projects on Windows and generate the result for Linux. As seen in the chapter **Introduction to .NET**, if you want to create a .NET Core application directly on a Linux operating system, you can install the Visual Studio Code without compatibility restrictions.

Since the 3.0 version, the .NET Core has included relevant performance improvements, allowing us to create low-memory applications and host our applications in simple containers on Linux. There are clear advantages of deploying and keeping .NET Core applications on Linux compared to Windows, the cost of licensing, and flexibility. Linux is an open-source project which can be customized for specific purposes. Additionally, it is also an alternative for high-secure servers, using **Security Enhanced Linux** (**SELinux**), which is included by default in many Linux distributions like Red Hat Enterprise Linux, Fedora, and a few extra ones.

In the example of this chapter, you will have the opportunity to deploy on a Linux server hosted on Azure, a web application built on Windows using .NET Core. First of all, you need to create the Asp.Net application by opening Visual Studio on Windows and choosing the Asp.Net Core Web application project type, as shown in *Figure 3.1*:

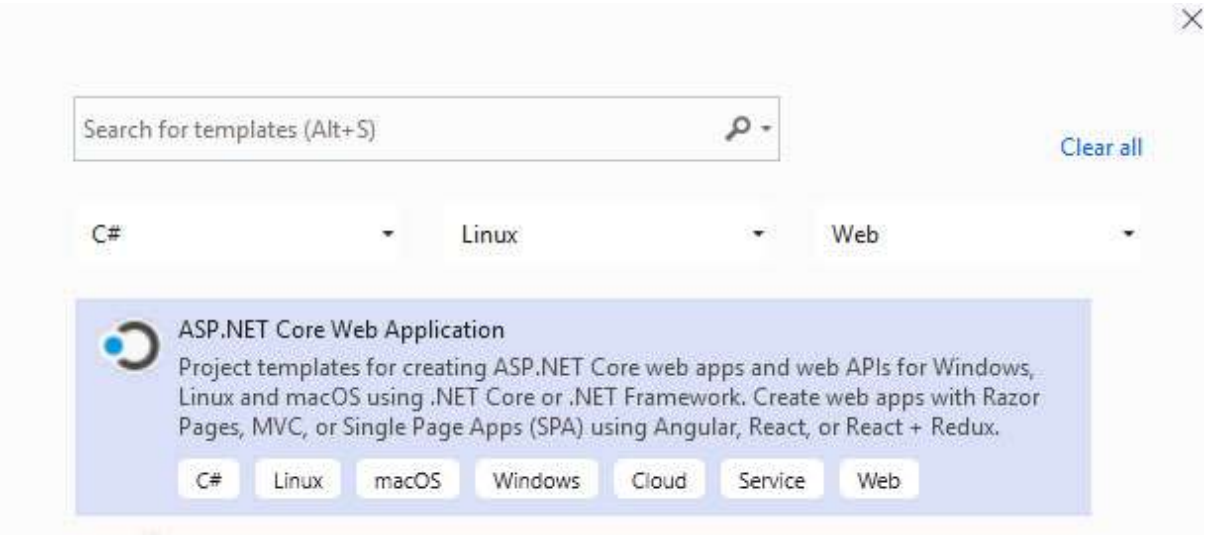


Figure 3.1: *Asp.Net Core Web Application for Linux*

It is possible to get the Asp.Net Core Web application by searching by text or even specifying the filters C#, Linux, and Web. As this type of project is compatible with Linux, it appears in the search results. After that, you must

choose the option Web Application, targeting the most recent Asp.Net Core 3.0 version, as shown in *Figure 3.2*:

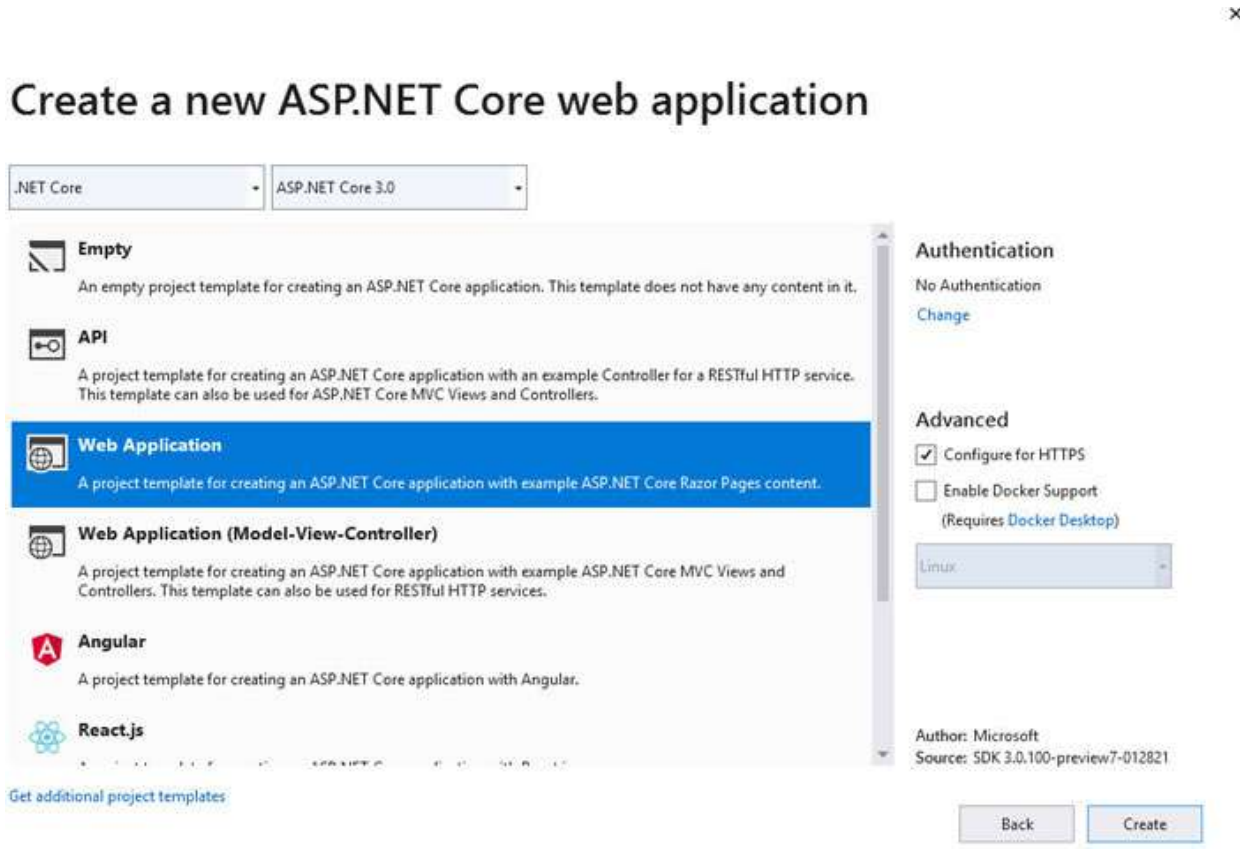


Figure 3.2: *Web application for Linux*

For deploying the Web application on a Linux server, this example uses the Azure App Service for deployment. The first step is to create a new App Service Plan on Azure. The App Service provides the infrastructure necessary to host and run your applications, and you can choose between Linux and Windows as the operating system.

Note: You need a Microsoft account to create resources on the Azure platform. Usually, Microsoft provides free accounts for students and free temporary accounts for tests. In any case, it is recommended to check all the available options online at the official website.

On the Azure portal, it is necessary to create a new resource by clicking on the “Create a resource” option, as shown in *Figure 3.3*:

Figure 3.3: *Create a resource option*

Once you have clicked on the option, you will be redirected to a search box where you can find every resource type available on Azure. In this example, you must choose the option "Web App," as shown in *Figure 3.4*:

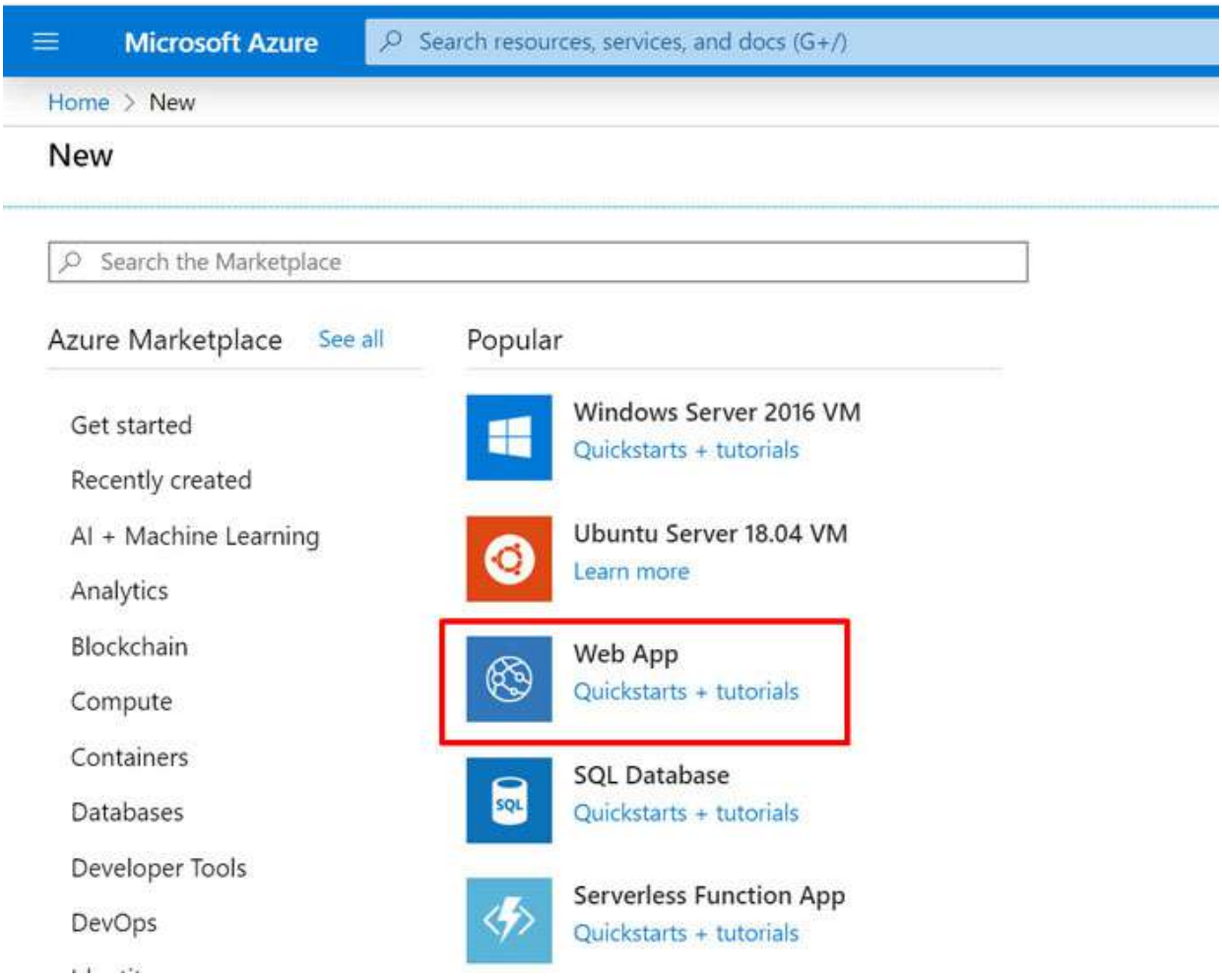


Figure 3.4: *Web app resource*

On the Web App resource form, you need to specify the following information:

- **Subscription:** With a single Azure account, it is possible to have multiple subscriptions, and each one will have separate billing management.
- **Resource group:** It is recommended to create a logical resource group to have all the resources that need to be controlled under the same purpose at the same place.
- **Instance details:** Each application on Azure should have a unique name and an exclusive domain. Therefore, that name should be specified and related to the Azure App service. Additionally, you can choose the operation system, the .NET version, region, and publish mode.

In this example, the following configurations were chosen, as shown in *Figure 3.5*:

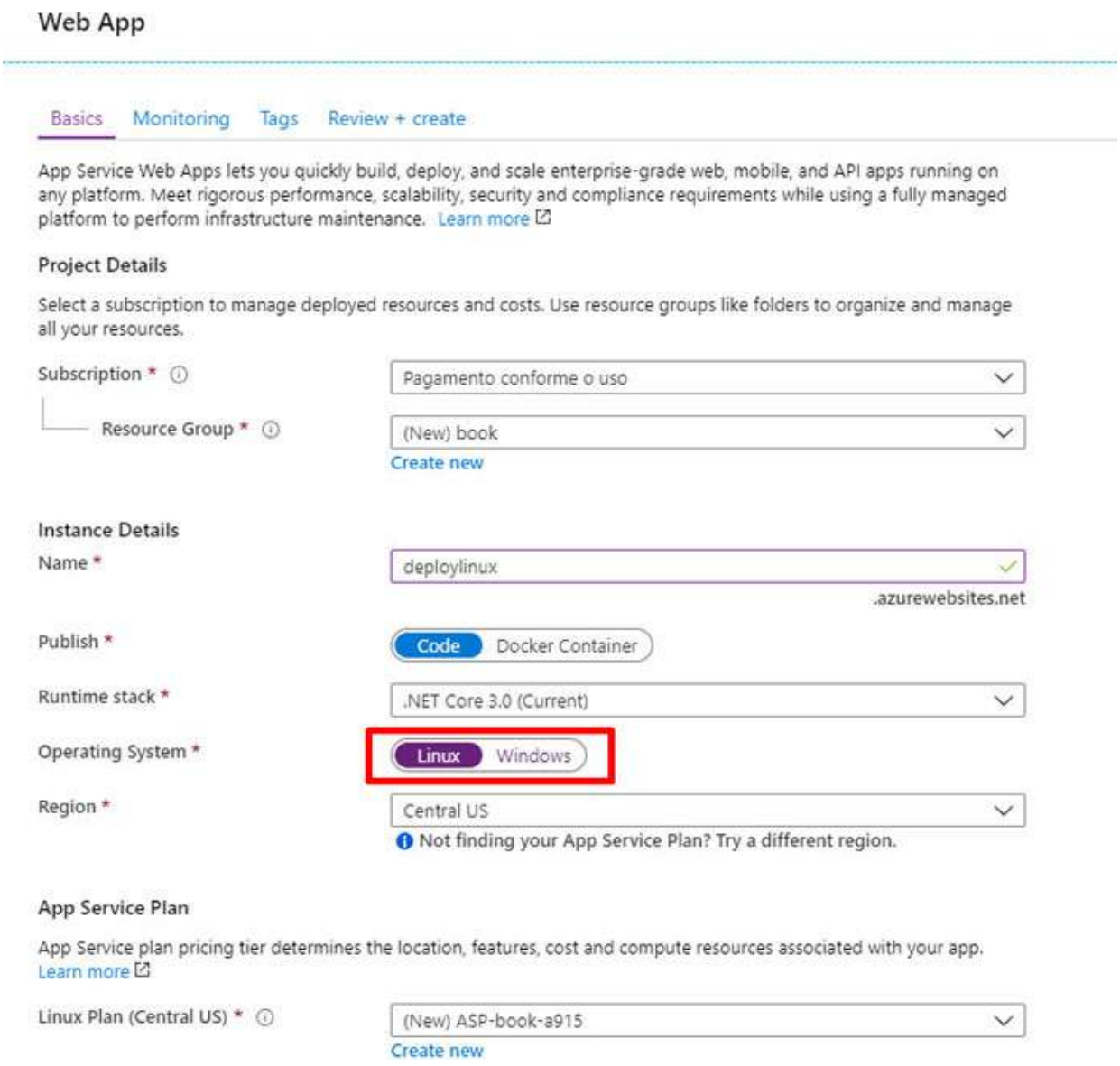


Figure 3.5: *Web app resource*

The Linux operation system was chosen, and the example application is named “deploying,” which means that it is the way to identify the application among other resources. It will have a unique sub-domain with the same name: deploylinux.azurewebsites.net. Using the free version of the App Service resource is not possible to use custom domains, but you can use your custom sub-domain for development and test purposes. Once the creation of the App Service is complete, you will receive an indication on the Azure portal, as shown in *Figure 3.6*:

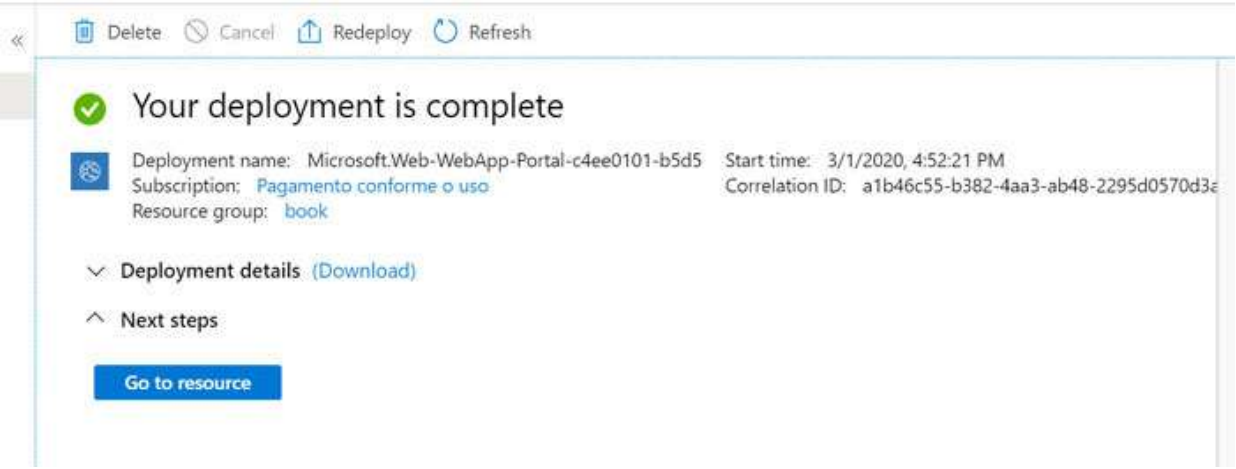


Figure 3.6: *Web app resource*

To effectively deploy the Asp.Net Core application on Azure Linux App Service, go to the Azure portal, open the resource you just created and change the HTTP protocol to “On.” Using secure encrypted communication between the application and the server is always recommended. After changing this configuration in TLS/SSL settings, the configuration will be shown as presented in *Figure 3.7*:

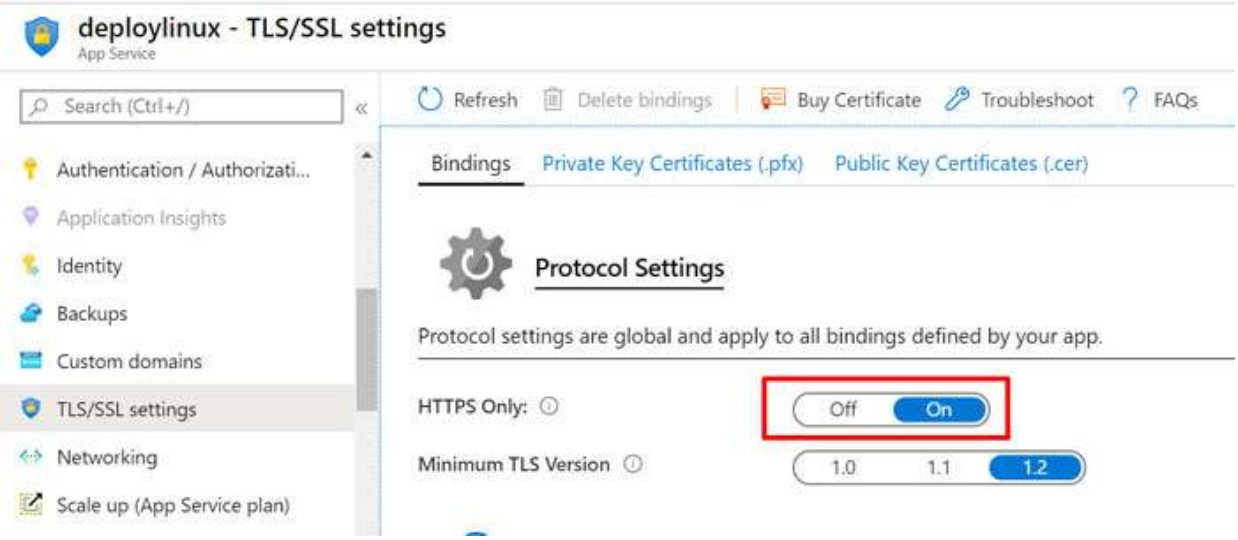


Figure 3.7: *Web app resource*

In the overview option, where you can see details of your App Service, click on the option “Get publish profile” and download the file that contains the necessary configuration to deploy your application using Visual Studio, as shown in *Figure 3.8*:

Figure 3.8: *Get publish profile*

After downloading the file, go to the Visual Studio instance where your application is open, click on the build menu option and choose the option "Publish," as shown in *Figure 3.9*:

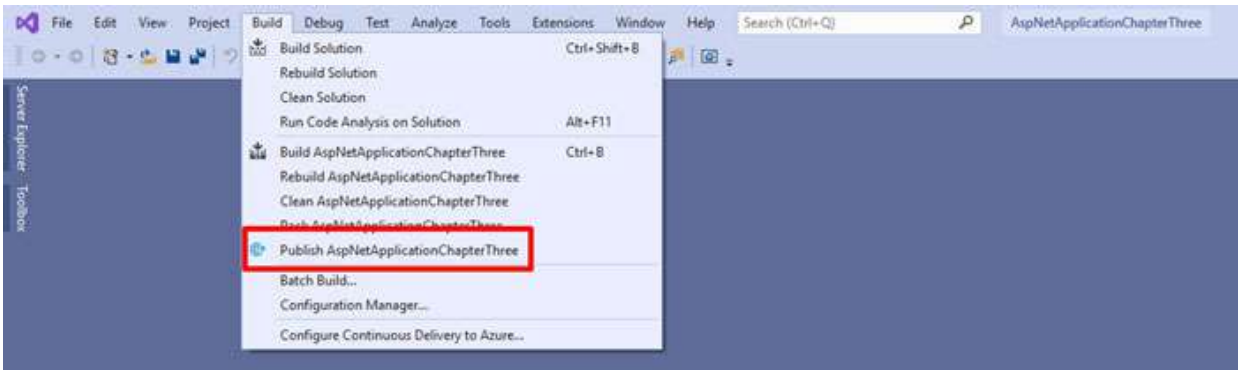


Figure 3.9: *Get publish profile*

Import the publish configuration file that was downloaded from Azure and click on the option "Publish" to deploy the application on the App Service based on Linux, as shown in *Figure 3.10*:

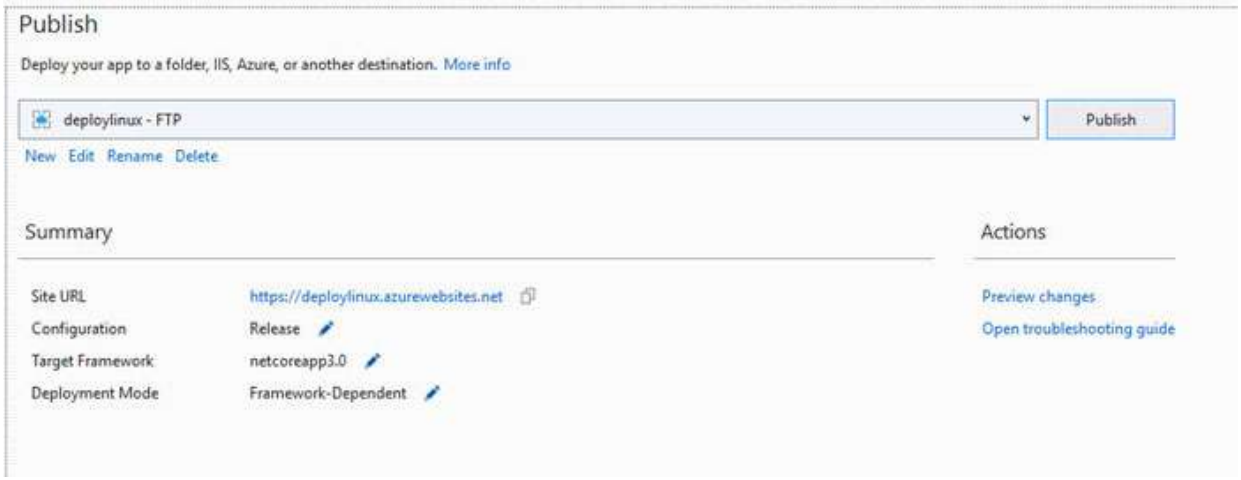


Figure 3.10: *Visual Studio publish option*

After deploying your application on Linux, Visual Studio will open an instance of your default browser pointing to the custom URL of the applications, as shown in *Figure 3.11*:

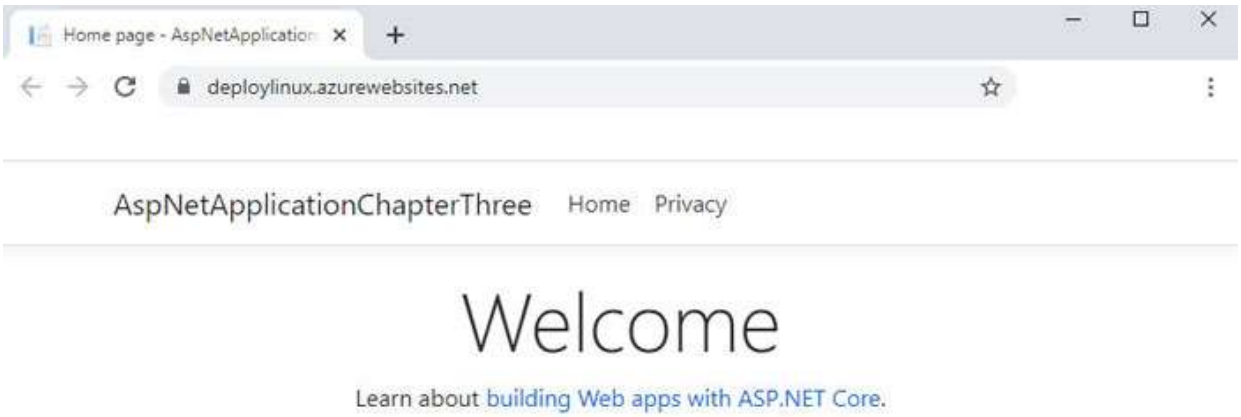


Figure 3.11: *Asp.Net Core application on Linux*

As you can see, with this deployment process, it is pretty simple to deploy Asp.Net Core applications on a Linux server hosted on Azure. Furthermore, this approach represents a cheaper way to host Web applications; considering it does not involve extra costs with licenses for operating systems. Therefore, it is a good alternative for low-cost projects.

Self-contained executables

The .NET Core 3.0 introduced the support for self-contained executables, enabling publishing and installing a .NET Core application as an individual executable file containing everything necessary to run on a specific platform. That facilitates the installation of applications on cross-platform environments since the client machine does not need any .NET Core installation or extra configurations, as the executable file has what is needed.

To test that functionality, you must create a Console Application on Visual Studio, choosing the underlying project on the Visual Studio project creation dialog, as shown in *Figure 3.12*:

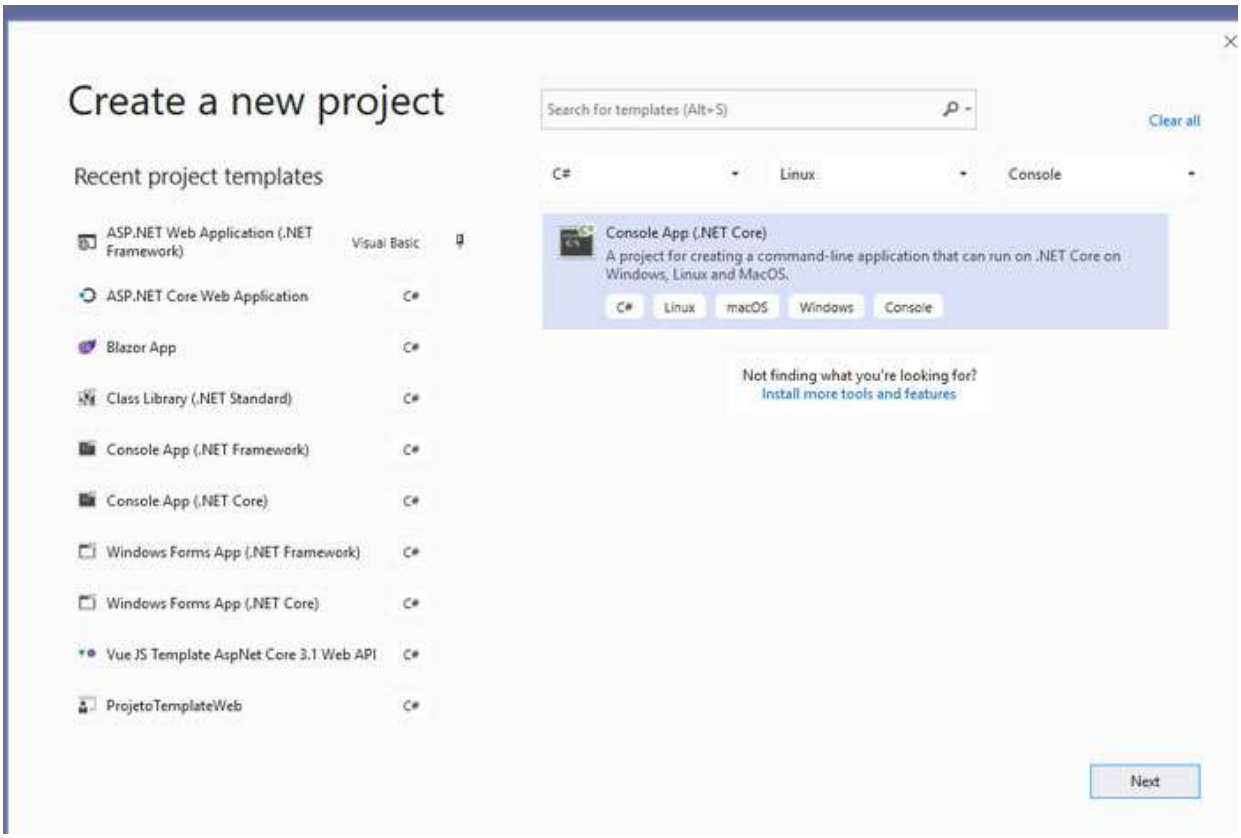


Figure 3.12: *Asp.Net Core application on Linux*

In the **`Program.cs`** file, replace the content with the code as shown in *Figure 3.13*:

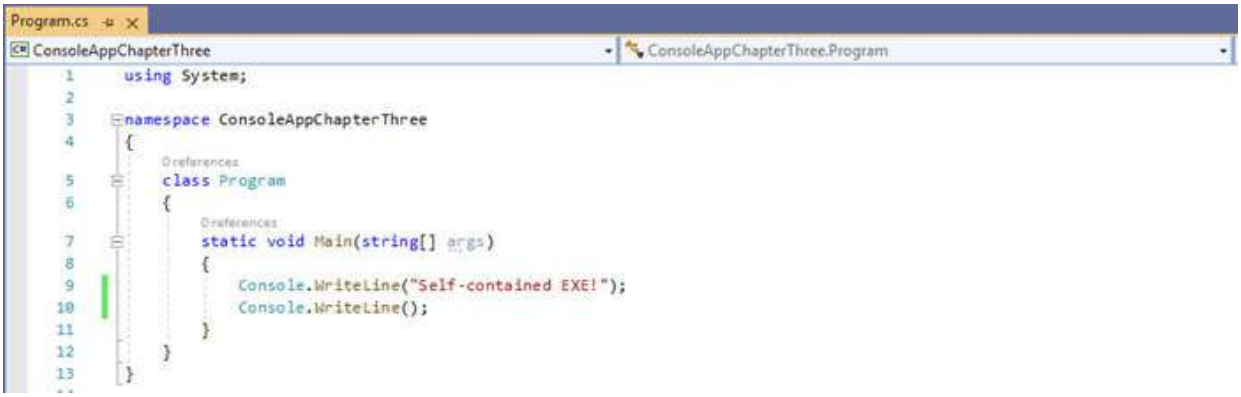


Figure 3.13: *Program.cs class*

This simple program shows the message “Self-contained EXE!” in the console and waits for a key in the prompt. As the purpose is to demonstrate how to deploy and prepare a desktop application on Linux, an extra configuration must be applied. There are two ways to generate a single

executable for Linux: one is to use the command line, and the other is to modify the configuration in the `.csproj` file. To change the project file, go to Solution Explorer, double click on the right mouse button on the project file, and choose the option “Edit Project File.” After that, replace the file content with the code, as shown in *Figure 3.14*:

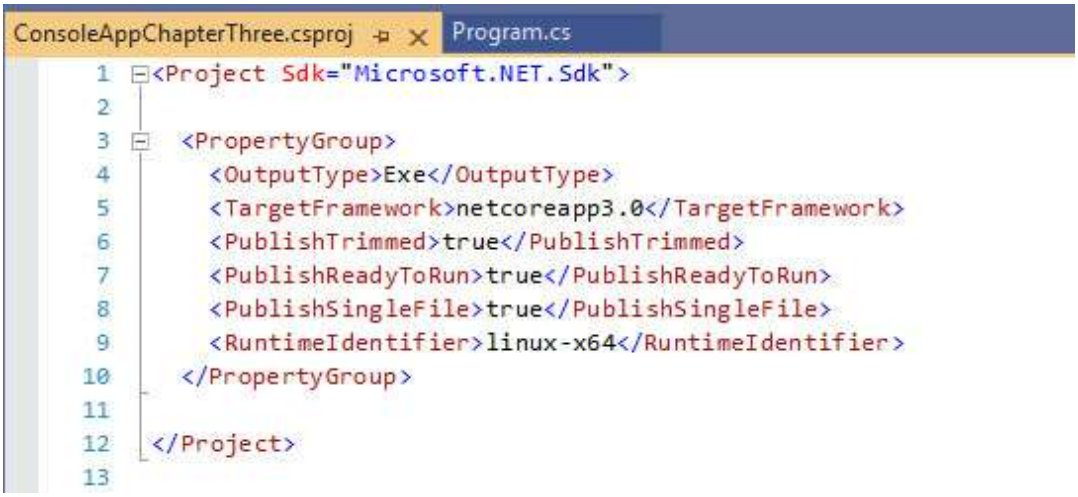


Figure 3.14: *Console App Project File*

In the runtime identifier tag, you can specify the operating system for which the final executable will be generated. You can also include the option to publish it as a single file. These options will build the application in a self-contained executable, ready to run on a Linux operating system.

After making these steps, change the build mode to “Release” and choose the option “Rebuild “on the Build menu, as shown in *Figure 3.15*:

Figure 3.15: *Release build*

If you go to the folder where your application was initially created, you will see in the release folder (inside the bin folder) that the executable was generated containing everything necessary to run and that it does not have any extra files, as shown in *Figure 3.16*:

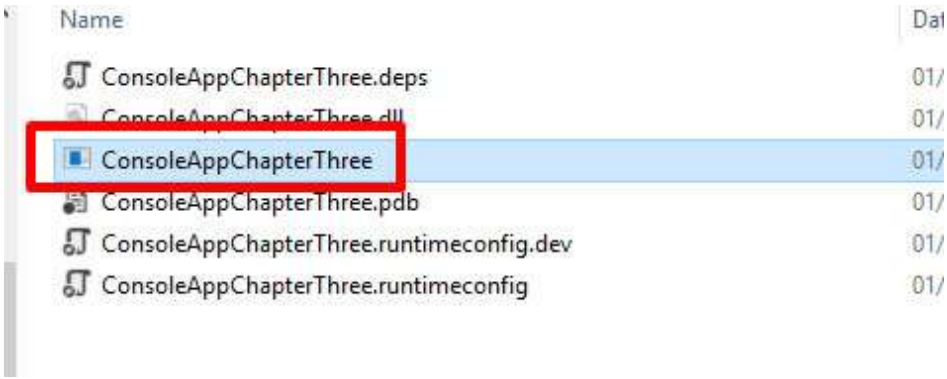


Figure 3.16: *Single executable*

This new feature was presented on .NET since .NET Core 3.0 reduced the cost of deployment in complex environments for desktop applications and facilitated the implementation of applications on Linux operation systems. In case the software needs to be installed and deployed on distinct operation systems, you can specify other types of platforms on the project file like it was done for Linux.

Conclusion

The .NET Core, since its first version, introduced many innovations in software development in terms of cross-platform architecture and compliance with the highest standards required for modern applications. With .NET, you can build robust enterprise applications and host them in any operating system and platform, reducing deployment costs and minimizing the complexity in scenarios where the same application must be deployed in an infrastructure that diverse companies provide.

As you can see in this chapter, since version 3.0, the .NET platform supports self-contained executables and gives more options to build desktop applications for Linux and other operating systems. Additionally, the process to deploy Asp.Net Core applications on Linux was simplified in the latest version of .NET Core, thus making it possible to make similar steps for building it on multiple platforms.

This chapter teaches you how to create and deploy Asp.Net Core applications on Linux, even after creating the application on the Windows operating system. Furthermore, you learned how to generate self-contained executables and the way to publish desktop applications in Linux.

In the next chapter, you will have the opportunity to get started with Object-Oriented programming using .NET and C#, doing practical exercises, and getting yourself familiar with the most basic concepts of the language.

Points to remember

- .NET Core applications can be hosted on Linux.
- Generally, a Linux server costs less than a Windows server and is a good alternative for .NET Core application deployment.
- It is possible to generate self-contained executables using .NET Core since version 3.0.

Multiple choice questions

1. **Which version of .NET supports self-contained executables?**
    a. .NET Framework
    b. .NET Core 2.2
    c. .NET Core 3.x
    d. .NET Core 1.0
2. **In which file is it possible to change the configuration to build executables for Linux operation systems?**
    a. Project file
    b. Solution Explorer
    c. Program.cs
    d. Startup.cs
3. **What is the recommended HTTPS protocol for secure Web applications?**
    a. Azure
    b. HTTPS
    c. Firewall
    d. HTTP

Answers

1. **c**
2. **a**
3. **b**

Questions

1. Explain the benefits of deploying .NET Core applications on Linux.
2. What are the advantages of using self-contained executables?

Key terms

- **Cross-platform applications**: Applications that can be developed and executed in any operating system: Linux, macOS, Windows, and others.
- **Self-contained executable**: An executable file that contains everything necessary to run the application, including extra DLLs, images, and configuration files.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

The Object-Oriented Programming

Introduction

Software development is an abstract process and correlation with the real world is done by implementing business or system requirements, which are represented in the software in a reliable way, in general, using the object-oriented programming paradigm. The concept behind that helped the software industry create more stable, readable, and extensible code for enterprise applications. It made the daily routine more accessible for developers, in general, bringing an intelligent design for any use developed with any language which supports that paradigm.

Beyond that, the object-oriented discussion brought to the market many additions and contributions, such as the SOLID principles concepts, which increased the quality of software across the world and gave the possibility to build robust and scalable applications, taking software engineering to the next level in terms of techniques and professionalism. Furthermore, because of all these facts, it is essential for any developer these days to learn this paradigm properly and to be able to apply its concepts in real and complex scenarios, which is a mandatory requirement for almost all positions in the market for developers.

In this chapter, you will have the opportunity to learn the essential concepts of object-oriented programming, such as classes, constructors, methods, abstract and static classes, inheritance, and interfaces. Additionally, explanations and examples of design patterns and SOLID principles will be introduced.

Structure

In this chapter, we will discuss the following topics:

- Classes, constructors, and methods
- Static classes and methods

- Structs
- Concepts of SOLID principles

Objectives

After studying this unit, you should be able to understand object-oriented programming concepts. We will discuss the essential aspects of design patterns and create fundamental applications using object-oriented programming. We will also learn how to use the SOLID principles in real projects

Classes, constructors, and methods

Starting from scratch with Oriented-Oriented Programming (OOP) concepts, you must know that any software in the market and any code implementation using a programming language has a target to achieve, an intention based on business or system requirements. There are many ways to solve a single problem in software engineering. Still, in terms of keeping and writing understandable code, it is vital to design code that represents the real-world situation related to the actual issue that the software is trying to solve. The OOP helps us abstract the complexity of software development and maintain a good correlation between technical implementations and most of the scenarios we can find in the market.

The C# language has supported OOP since its first version, and every application that uses .NET Core must apply this paradigm; considering all the native libraries of the platform are based on this concept. Additionally, the .NET platform takes the benefit of advanced concepts of design patterns and SOLID principles, which will be explained later in this book, using real-scenario examples.

The OOP allows us to organize the implementation of software requirements around objects and represent the behavior and state of something that needs to be abstracted in the system. An object can contain properties and methods which will be responsible for keeping and manipulating its state and certain operations related to the object itself. In C# language and other ones based on the OOP paradigm, an object is represented by a class that should have similar characteristics to the real world in a faithful representation.

For better understanding, just imagine a scenario where an online store needs to be developed using C# language and .NET Core. According to the fictional case, the system should implement the following requirements:

- Allow users to register products and product types.
- Every product should have a unique number to identify the product and a title.
- Users access the online shop but don't necessarily make a purchase.
- Once a user makes a purchase in the online shop, the same user should be registered as a customer with additional information.
- The products can contain different properties based on their type.

Considering the given scenario, if we would start the plan of the structure of our code to keep everything well organized, we must create the classes first, with the basic properties that are common to every object of these classes. Starting from the model for the product, the initial implementation of the class would have the implementation as shown in *Figure 4.1*:

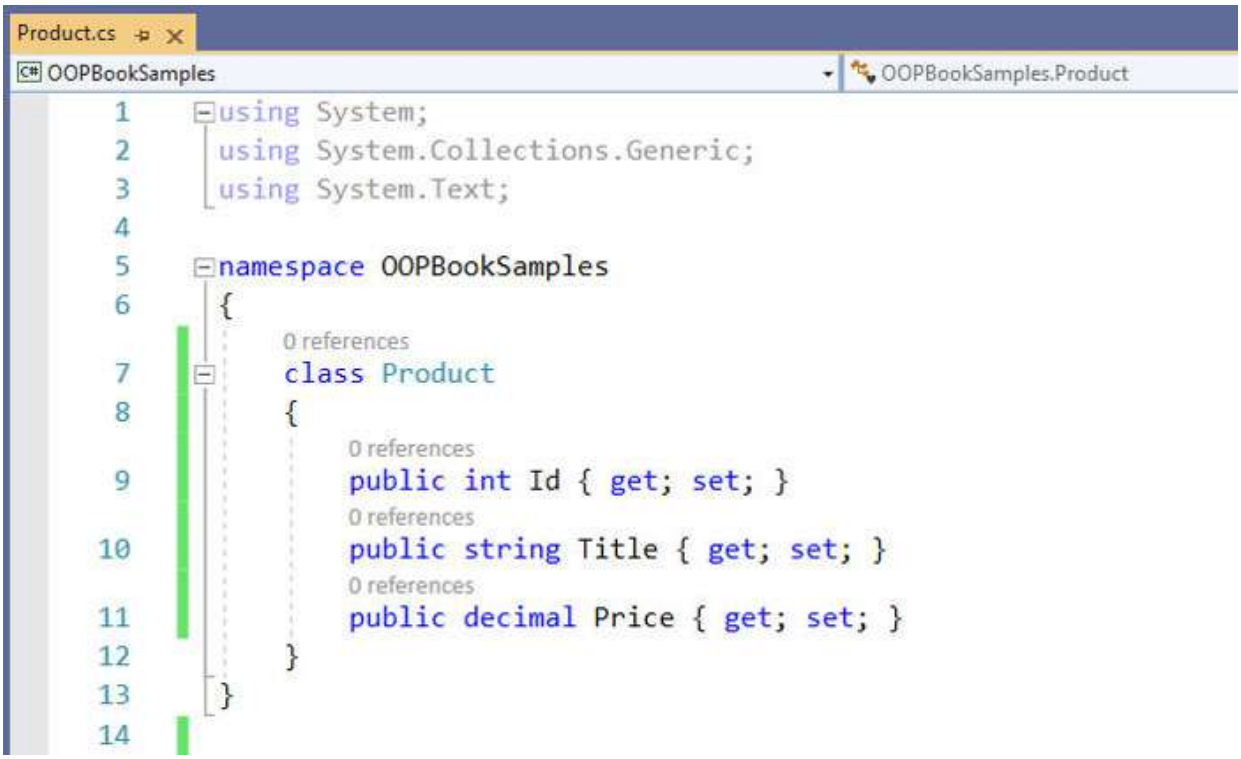


Figure 4.1: *Product class*

In that case, the fields ID, Title, and Price represent the properties of the Product class, and each one has its own type, such as an integer, a string, and a decimal. You must realize that there is a keyword before each property. The keyword means the access modifier or scope of the property. And in OOP, the following are access modifiers:

- **Public:** The class or property can be freely accessed by other classes; even those not present in the same assembly.
- **Private**: The property can be accessed only by the same class to which the property belongs.
- **Protected:** The property can be accessed only by the same class or the derived classes.
- **Internal:** The property or class can be accessed by other classes, but with the limitation that the class should belong to the same assembly.
- **Protected internal:** The class or property can be accessed by any other class in the same assembly and for all derived classes, even if they are presented in another assembly.

Depending on the keyword, the access level changes, as shown in *Figure 4.2*:

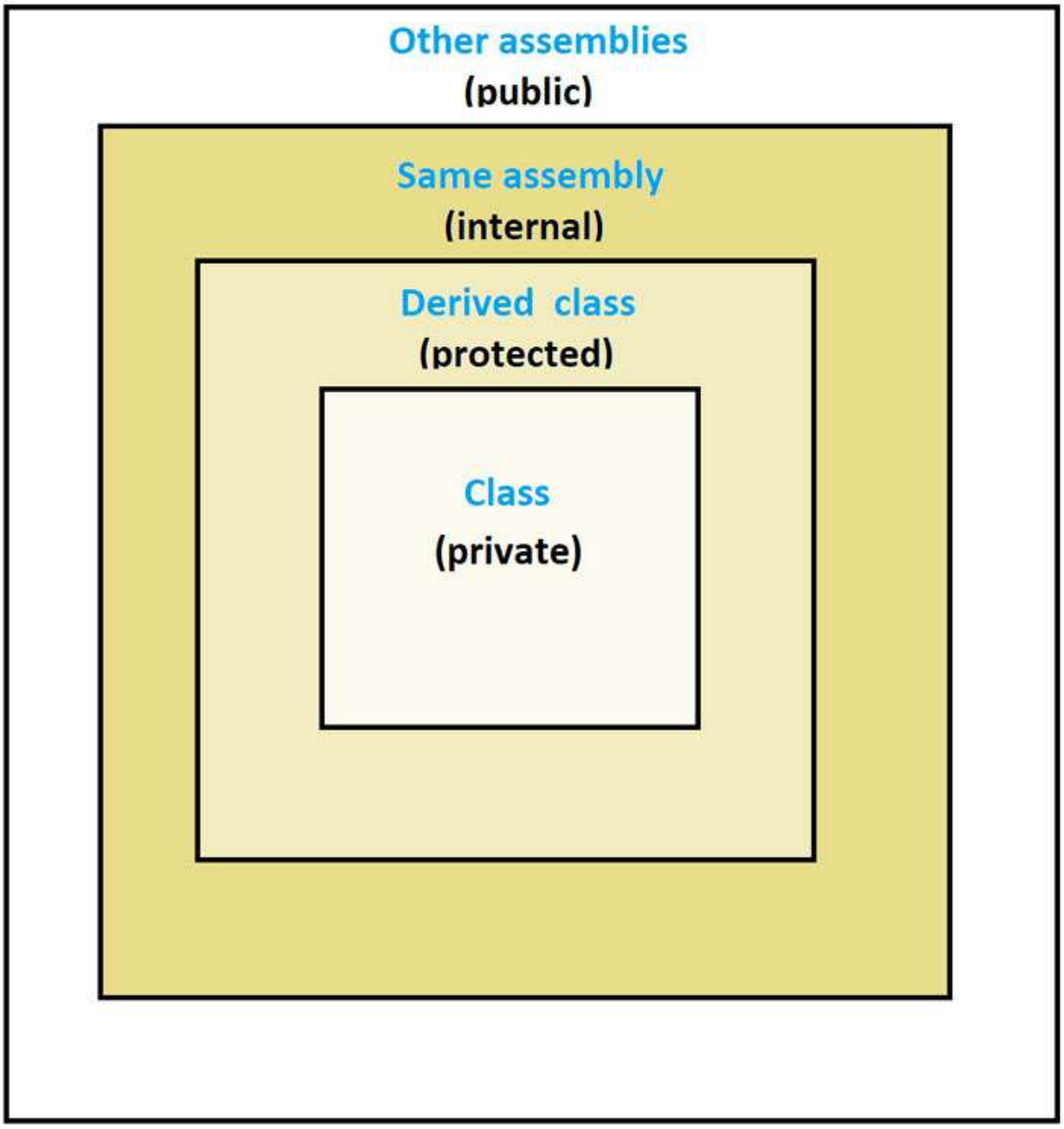


Figure 4.2: *Access level*

Object-oriented programming has principles that enforce the best practices of coding using this paradigm and has the purpose of not violating the mechanisms that should be followed in any software that uses these concepts, such as encapsulation, inheritance, polymorphism, and reusability. The following sections have detailed information on each of them.

Encapsulation

This concept aims to avoid external interference on the object state and protect access to the properties of the class and its methods. To meet this objective successfully, it is essential to understand the access level property, which was explained earlier in this chapter. Control the access to properties, methods, and classes to ensure that only the code that needs access to other specific codes is under control. Considering all the classes in a system should be low coupled with each other, which means that as much as possible, each class must be independent and contain the necessary code to meet its target.

Inheritance

Following the principle of reusability, it is possible in object-oriented programming to use the concept of inheritance, which allows us to share properties and methods that mostly have something in common. A class that inherits from another class is called a derived class, and the base class is called a parent class. The C# language allows us to use only one inheritance associated with a class, but it is possible to use interfaces to meet a similar objective. Moving forward with the previous example related to the online store, a store can contain many types of specific products that would share common properties and methods between them, but other properties might be specific for each one of those. For instance, the class “Movie” is a type of product that shares similar characteristics to other products, such as ID, Title, and Price. But specific properties are only related to the Movie class. Considering the class product as shown in Figure 4.1 already contains particular properties, the Movie class can inherit from the Product class, as shown in *Figure 4.3*:

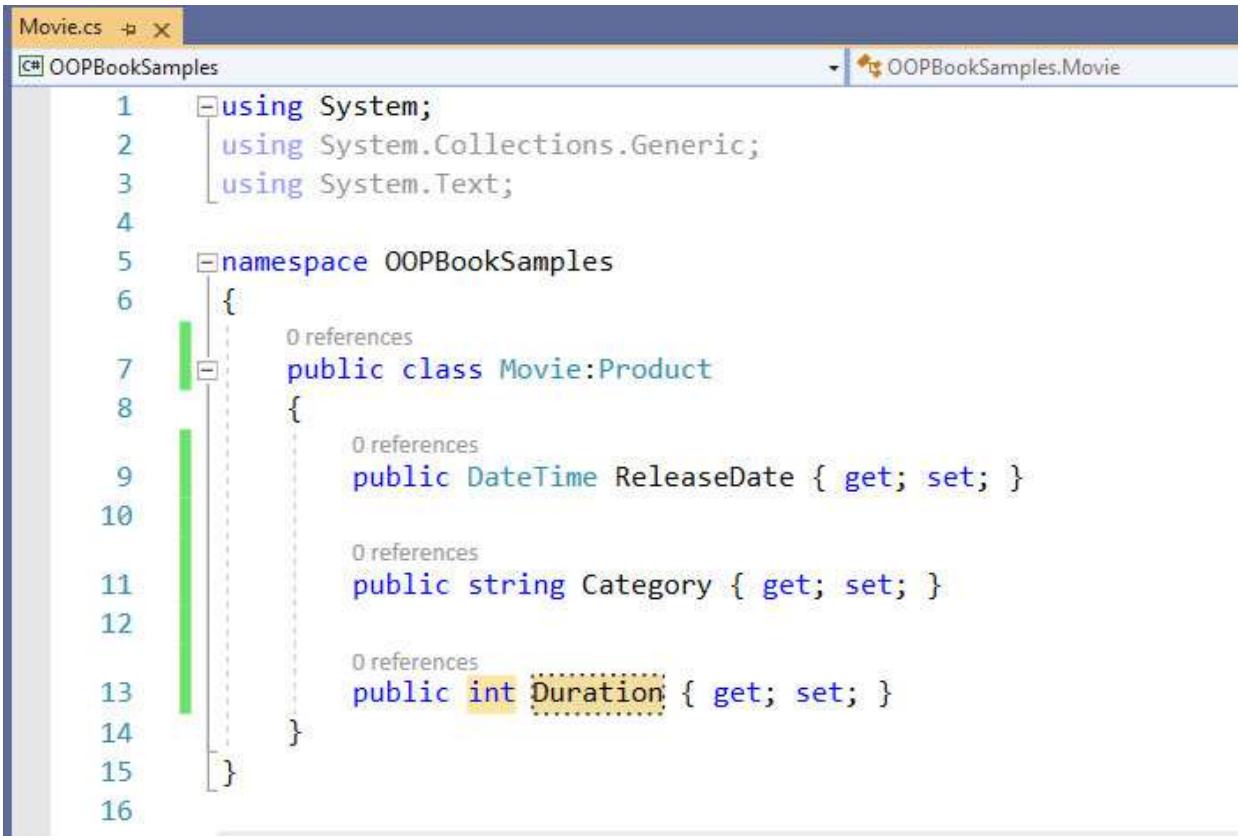


Figure 4.3: *Movie class*

In the signature of the class, inheritance is declared from the Product class, which means the Movie class will contain all the properties from the Product class and the specific fields from its class: release date, category, and duration. The decision to inherit or not from another class depends on the context and generally is based on business requirements. The inheritance concept can be suitable if different classes have a clear correlation. This practice helps developers save time writing code across the system and keep the system architecture consistent.

Reusability

Generally, the costs of any software are measured by the number of hours spent writing the code, among other activities such as business analyses, software design, and management. Considering the time to develop software represents one of the essential points for a project to succeed, it is necessary to use techniques that allow us to save time in building the software we are

working on. Reusability is an essential concept of object-oriented programming because it represents less time to develop systems.

Going back to the online store example, considering it has already implemented a base Product class, if the online store had a vast number of products of different types, using the base class would save relevant time in implementing the other classes. It also applies to more complex scenarios where simple properties and methods can be reused and whole libraries and external functionalities.

Polymorphism

This concept in objected oriented programming allows us to overwrite a method from a parent class and share among multiple classes a standard interface to give integrity and consistency in the software architecture. As its name suggests, polymorphism is the possibility of having many forms for a common method. In the online store example, imagine that each product could have its way of applying discounts based on custom rules. In that case, the base class (Product) would have a standard method to calculate the discount, but all the derived classes can overwrite their behavior according to the specific situations.

For instance, the parent class (Product) can have a specific method for applying a seven percent discount, as shown in *Figure 4.4*:

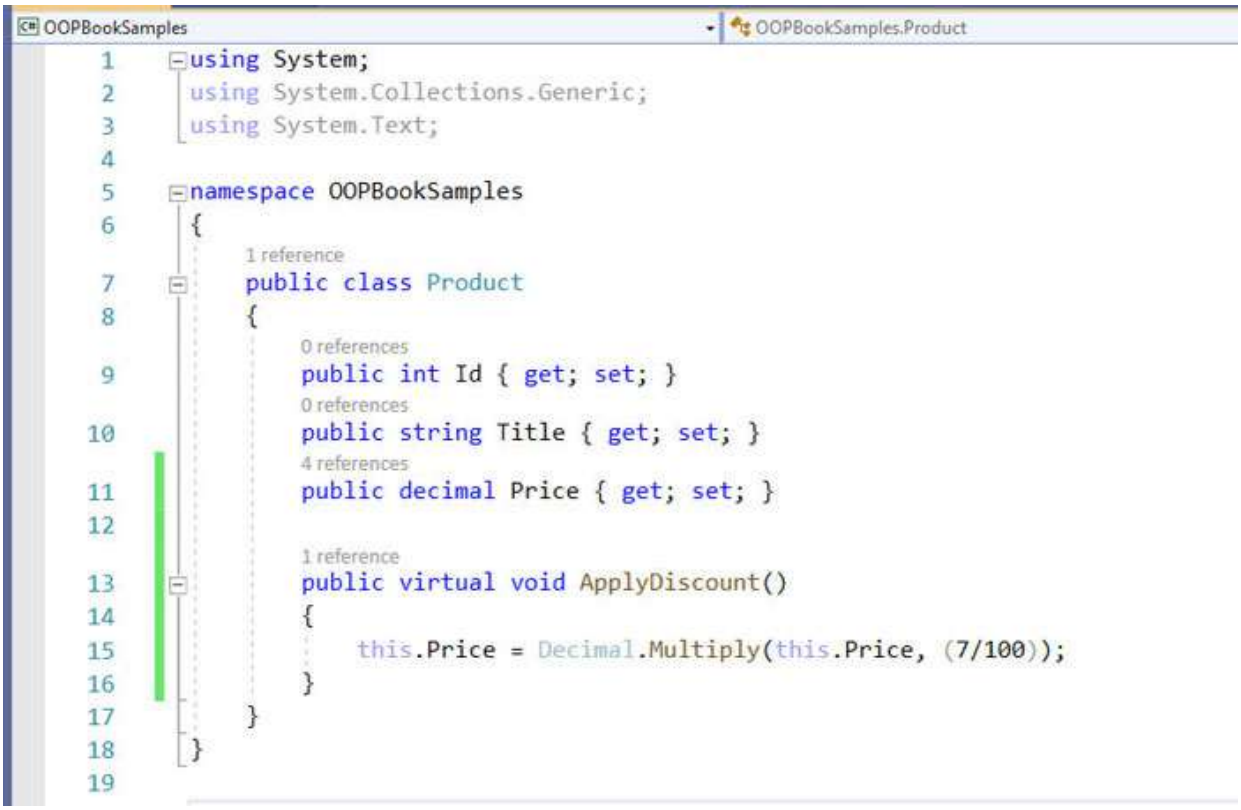


Figure 4.4: *Method to apply the discount in the Product class*

But, on the other hand, the child class (Movie) can overwrite the implementation of the **`ApplyDiscount`** method, as shown in *Figure 4.5*:

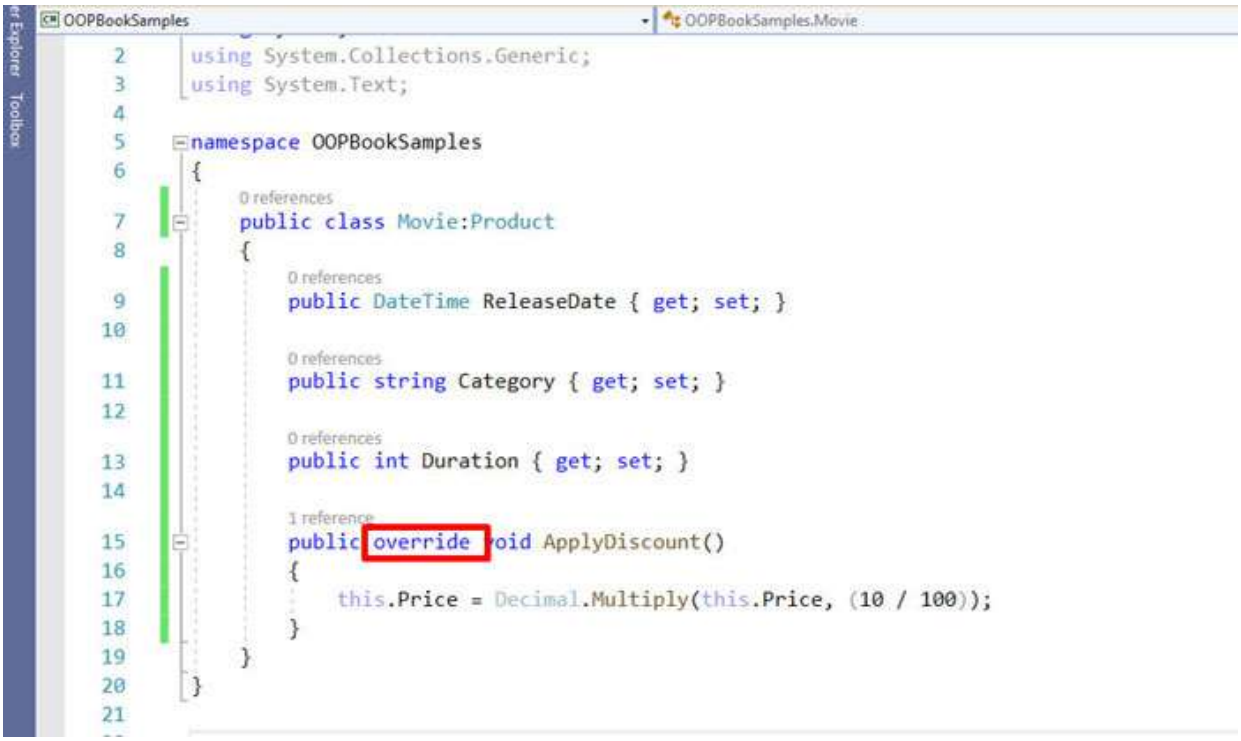


Figure 4.5: *Method to apply the discount in the Movie class*

The Movie class applies a different discount policy once the discount value is 10 percent, therefore distinct from the parent class. It is possible to change the implementation of a method from the parent class only if the method in the parent class has the modifier “virtual,” as shown in *Figure 4.4*.

Partial class

To keep the good practices of coding and project structure, usually, all the classes are presented in the same file, which will have the same name as the class, as highlighted in *Figure 4.6*:

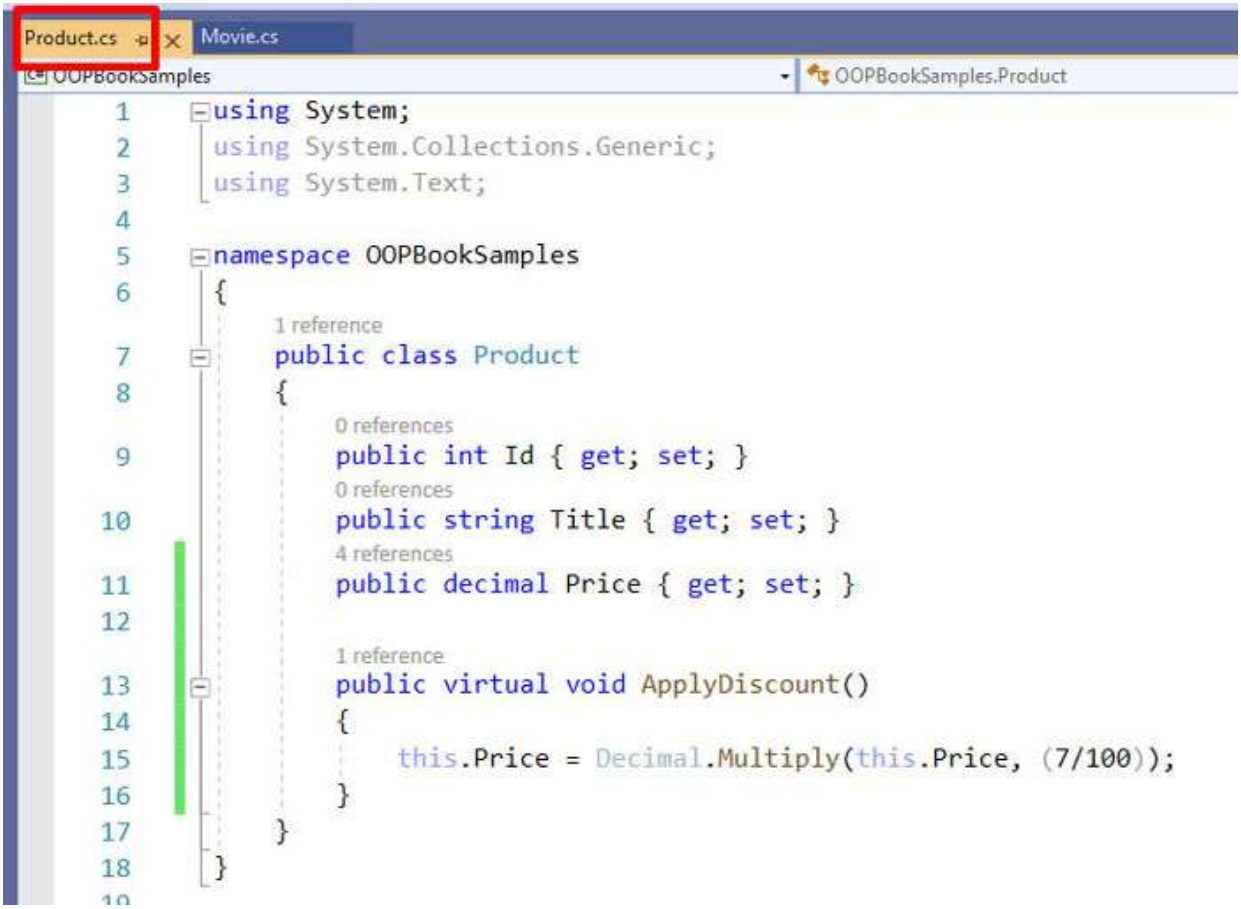


Figure 4.6: *File name pattern for classes*

However, if the class contains a significant number of code lines and in the situation where many developers need to work at the same time in the same file, it is convenient to split the class into multiple files to avoid conflicts and to have a logical separation for a better understanding. Even though the class is presented in more than one file, the compilation will combine all of them as a single class. In the following code example, there are two files regarding the Product class: one containing the fields and another one containing the methods, as shown in *Figure 4.7*:

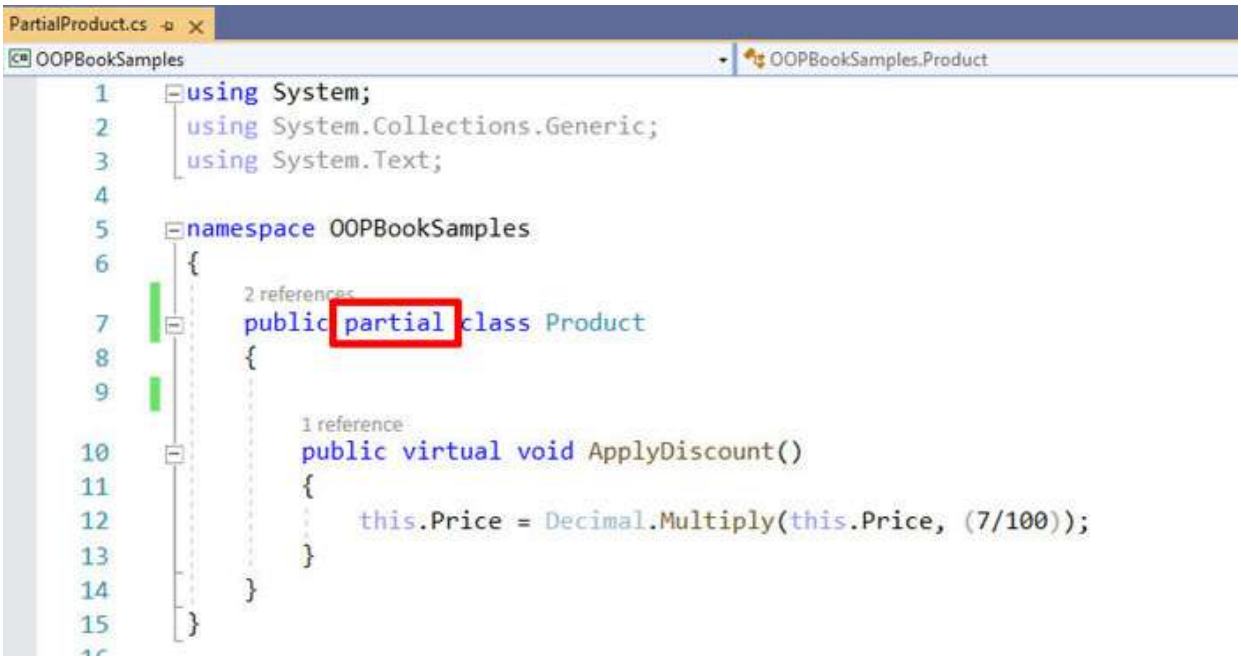


Figure 4.7: *Product partial class*

The keyword **`partial`** was used to transform the class into a partial class, containing only the apply discount method. On the other hand, the other file that includes the implementation of the Product class will only provide the properties, as shown in *Figure 4.8*:

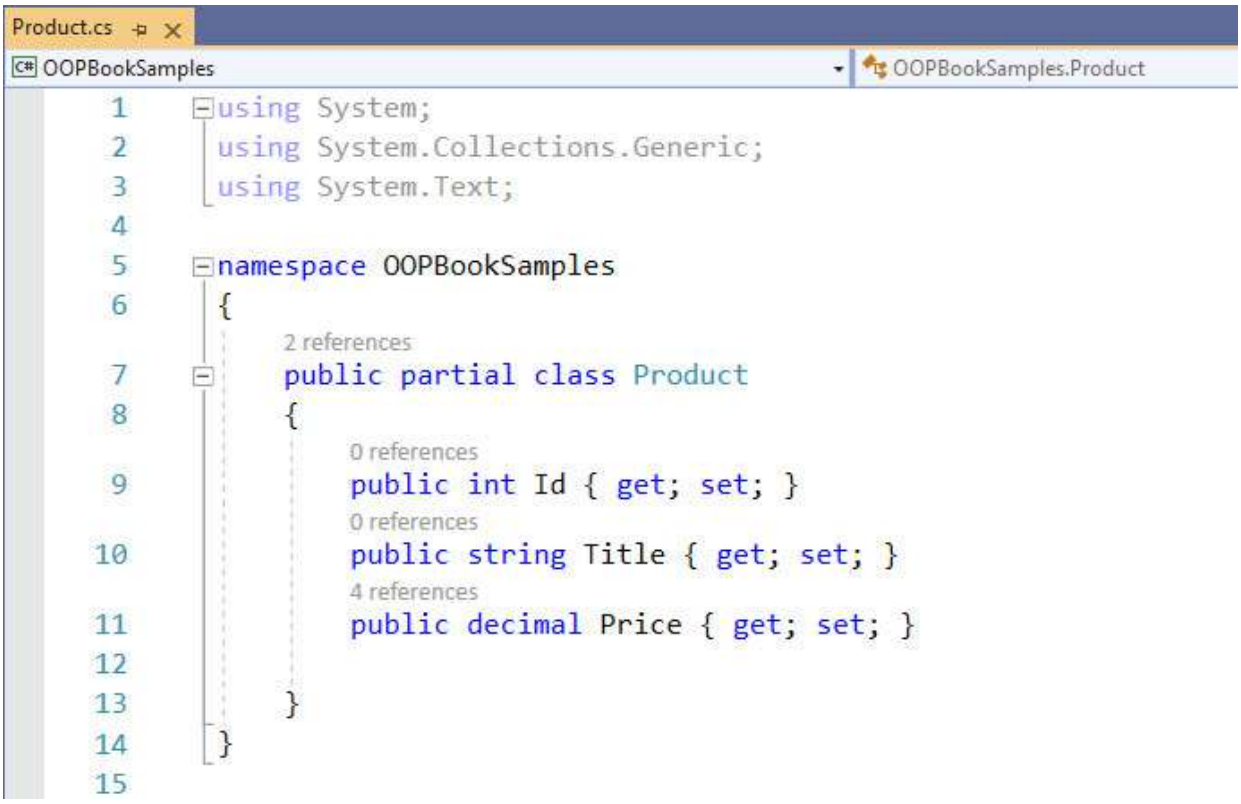


Figure 4.8: *Product partial class*

The use of partial classes is quite flexible, but it is recommended to do that just in cases where the class is big enough to be split. Additionally, if a class contains a lot of lines of code, usually that is a clear indication that the class should be refactored to follow the single responsibility principle, which is a good practice stated by the SOLID principles that will be explained later in this chapter.

Constructor

In object-oriented programming, the constructor is the method called when an instance of the object is created. In other words, a class contains the implementation, and every time a class is used, an object is created having the same type as the class. In the scenario of the online store, considering there is a class called Product in the system, every time this class needs to be used, we must create a new instance of an object referring to the class, as shown in *Figure 4.9*:

Figure 4.9: *Product object instance*

When the keyword **new** is used, it indicates to the compiler that a new instance of the Product class should be created, and this object will be placed in a new space in the memory. Additionally, the new object is initialized in this operation, and the constructor method is called. You can specify many constructor methods in the class, and all of them should have the same name as the class, as shown in *Figure 4.10*:

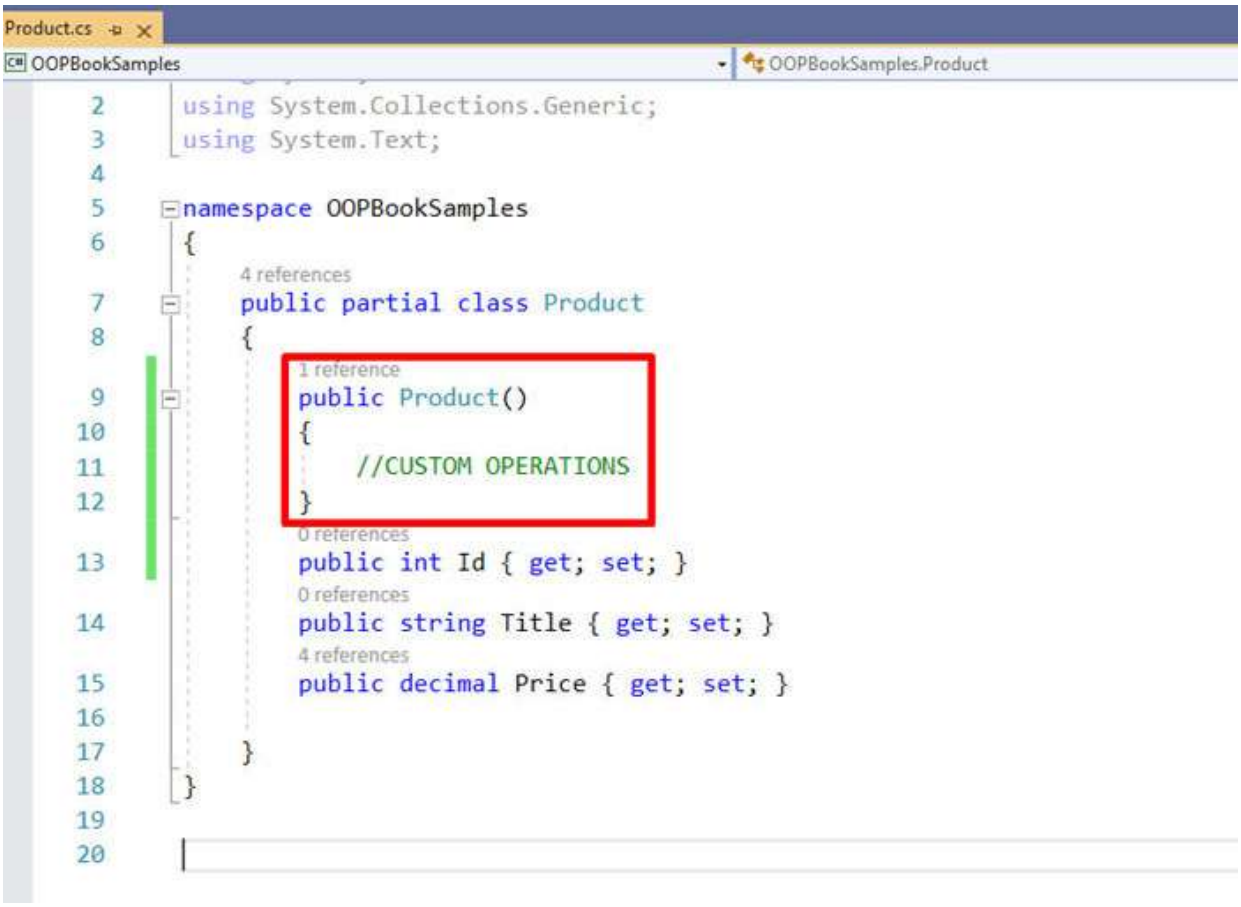


Figure 4.10: *Constructor*

Usually, the constructor methods are used to set the initial state of the object, and if the constructor method is not specified, the compile will run the default constructor method, which is an empty method. It is essential to understand how constructors work because the state management of the objects is one of the main points to applying the object-oriented programming paradigm's best practices correctly.

It is possible as well to pass parameters in the constructor, which in specific cases are necessary to make operations when an object is initialized, as shown in *Figure 4.11*:

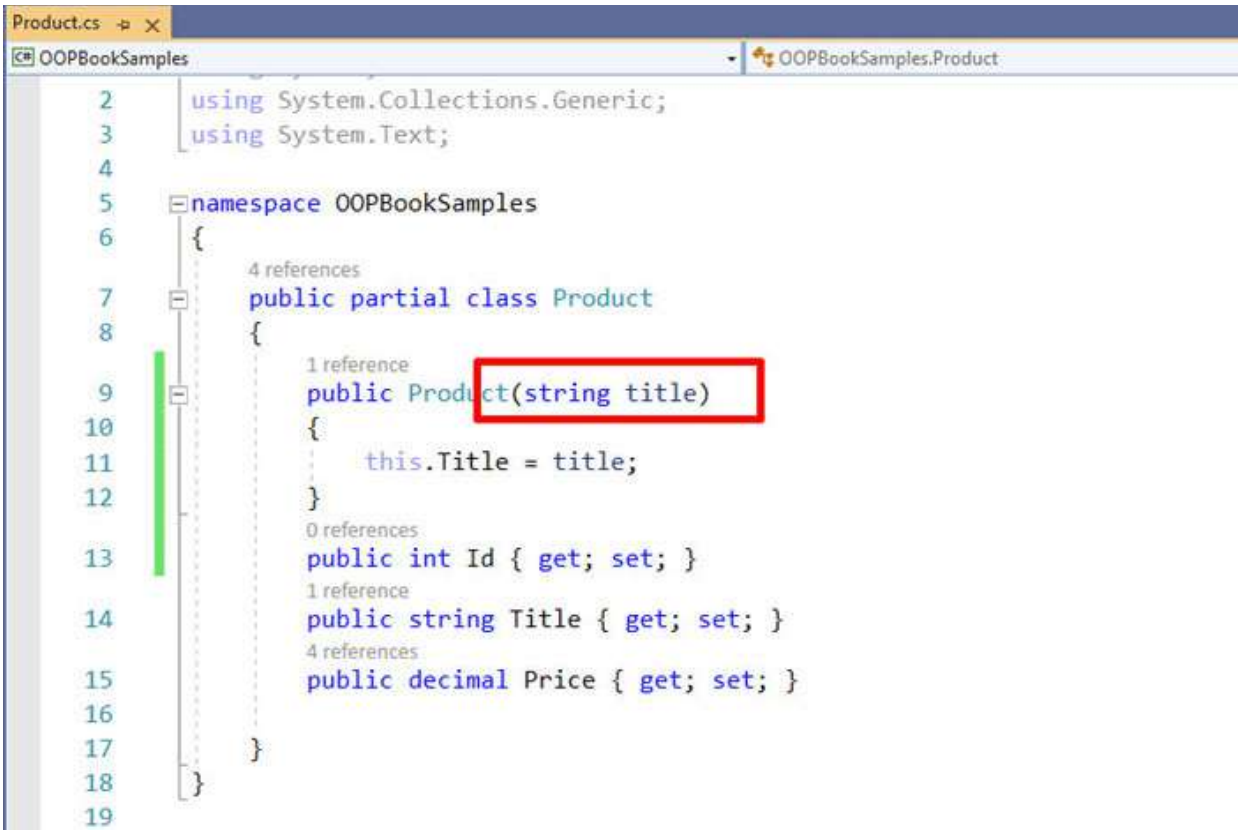


Figure 4.11: *Constructor with parameters*

Static classes and methods

In C# language, a static class represents a class that cannot have an instance created. That means the system will keep a single class instance in the memory, which will be used for the whole system. Considering that characteristic, it is not possible to use the keyword new when there is the intention to use the static class. Usually, the static classes are used in cases where the class does not need to keep its state dynamically. For instance, the online store can have a static class to make certain common operations for the system, such as type conversions, math operations, and others. The following figure contains an example of a static class called Helper, which has a method that returns a unique identifier:

Figure 4.12: *Static class*

As highlighted in the preceding figure, the keyword static was used to transform the class Helper into a static class. Considering it is not possible to create an instance of static classes, the compile will generate an error if an implementation tries to proceed with that, as shown in *Figure 4.13*:

Figure 4.13: *compile error for static class*

To use the methods presented in the static class, you must call the methods directly without using the keyword new, as shown in *Figure 4.14*:

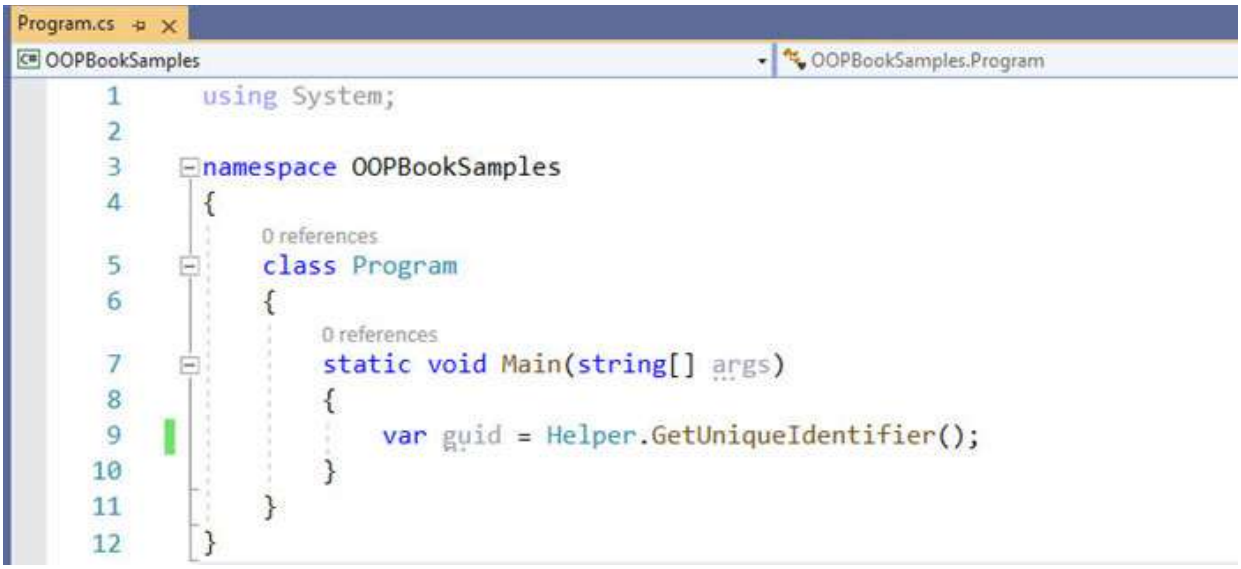


Figure 4.14: *Correct use of static class*

Structs

The structs in C# language have a similar implementation to classes, including properties and methods. However, the main difference between them is how they are kept in memory in a program developed in C#. The conventional class is considered a reference type, which means that the type does not store its value in the memory but only the address where it will be stored. On the other hand, structs contain their own value, and because of that, access to their values is much faster than reference types. Furthermore, structs have restrictions in comparison to classes, as follows:

- It is not possible to use structs in an inheritance process.
- It is not possible to use an empty constructor. It is mandatory to have a constructor with parameters.
- It is required to set values to local variables when they are created.
- The instance of the object created from a struct class is removed from the memory once the method that is using the object has its process completed.

The syntax of the struct is quite similar to the class, as shown in *Figure 4.15*:

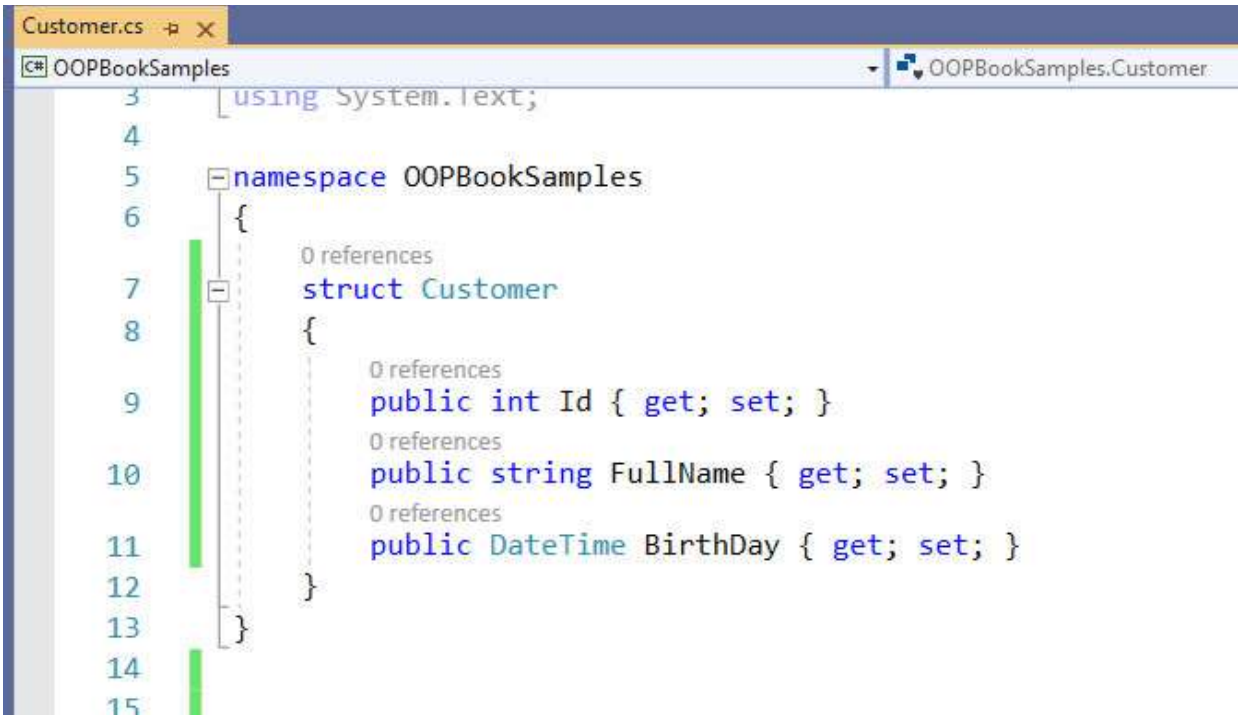


Figure 4.15: *Struct class*

A struct must have a constructor with parameters, as shown in *Figure 4.16*:

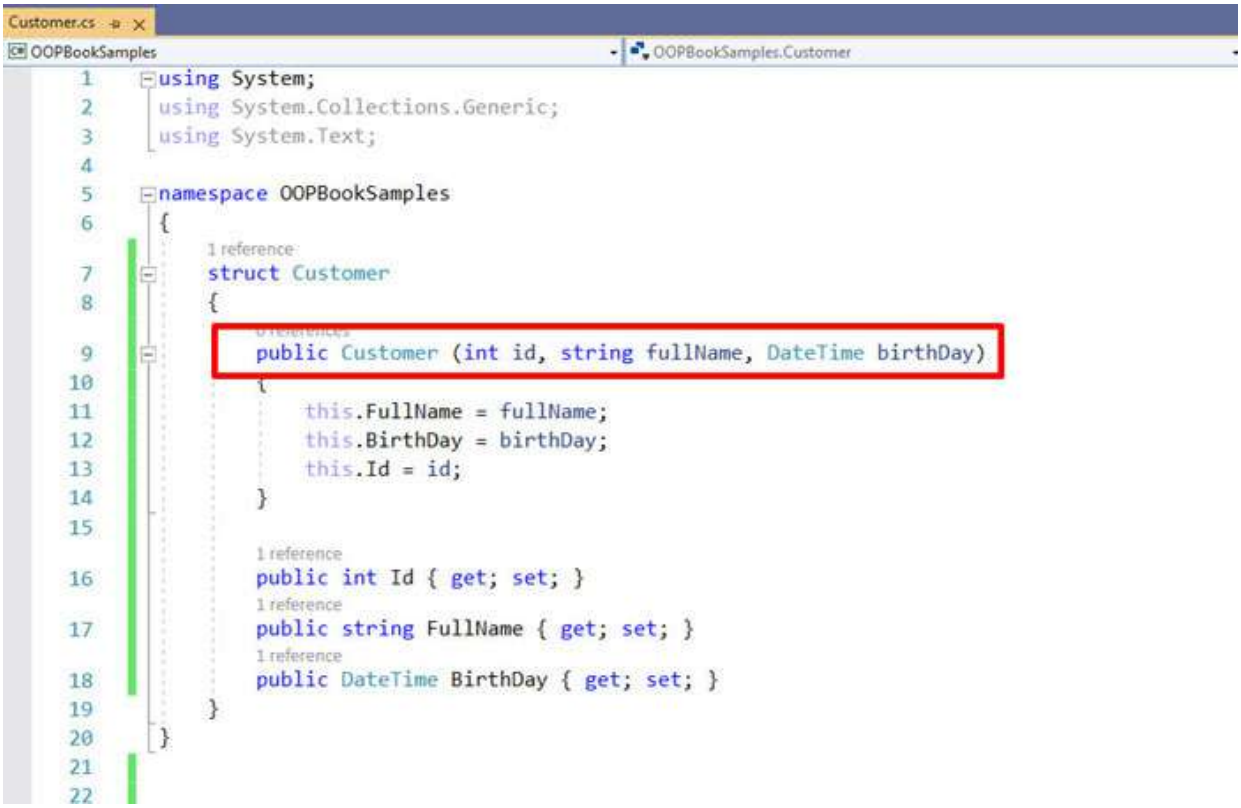


Figure 4.16: *Constructor for struct*

The use of structs is pretty helpful in the cases of a specific part of the system that requires high performance and when the primary purpose of the implementation is to only keep a temporary state of the object, with known end-of-life inside the method calls.

Interfaces

In object-oriented programming, an interface represents a contract that should be followed by all the classes that implement the interface. In other words, an interface contains everything that a class must have, such as properties and methods, following specific signatures. This concept is significant for correctly implementing business requirements and consistency in the development process.

In C# language, an interface can be created with the interface keyword just before the interface's name. As a concept, the interfaces don't contain any implementation, which is the responsibility of the classes that will implement those. Going back to the online store example, considering there would be a requirement that all the product types should implement a method reference to delivery, an interface can be created to ensure that all the classes related to product type will implement this method. A primary interface for the product would appear as shown in *Figure 4.17*:

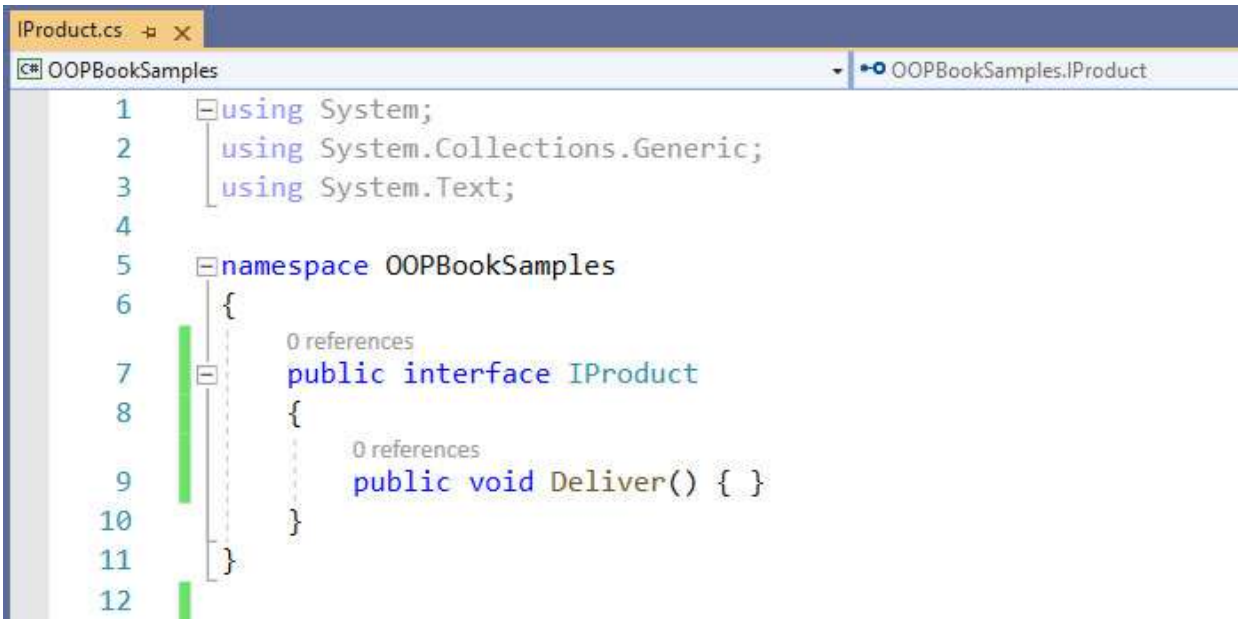


Figure 4.17: *Constructor for struct*

According to this interface, it is mandatory for all the classes that will implement this interface to have a method called “Deliver” with the same signature without any return. To enforce the contract between the interface and related classes, in the signature of the class, you must declare it after the name of one or more interfaces, as shown in *Figure 4.18*:

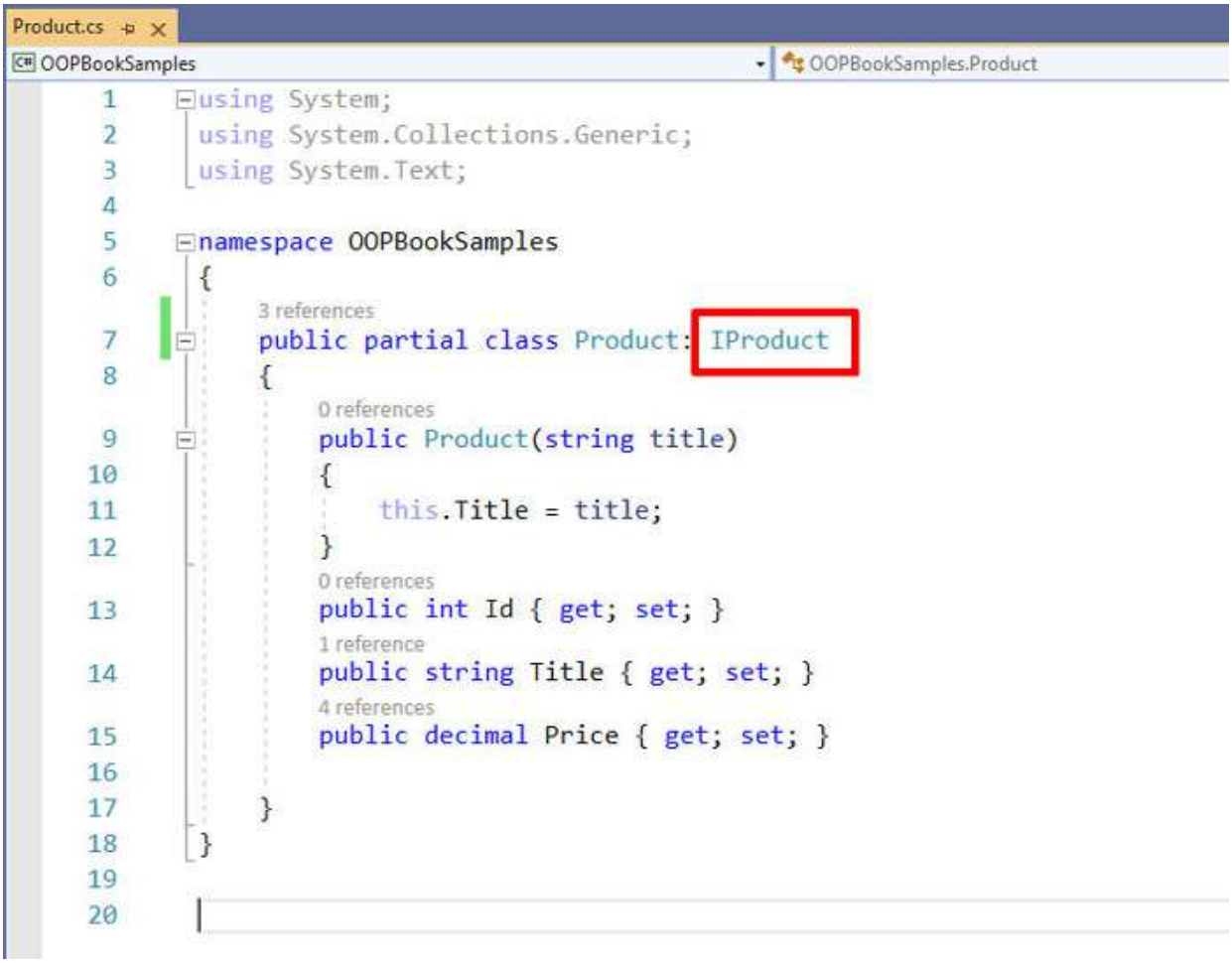


Figure 4.18: *Constructor for struct*

Since the implementation of the delivery method is mandatory for the Product class from that point, before version 8.0 of C#, the compile would show an error accusing that the underlying implementation is missing. But since version 8.0 of the C# language, a default implementation can be specified in the interface, and the related classes don’t need to implement the method with the same signature. Therefore, it is possible to have a default implementation in interfaces, as shown in *Figure 4.19*:

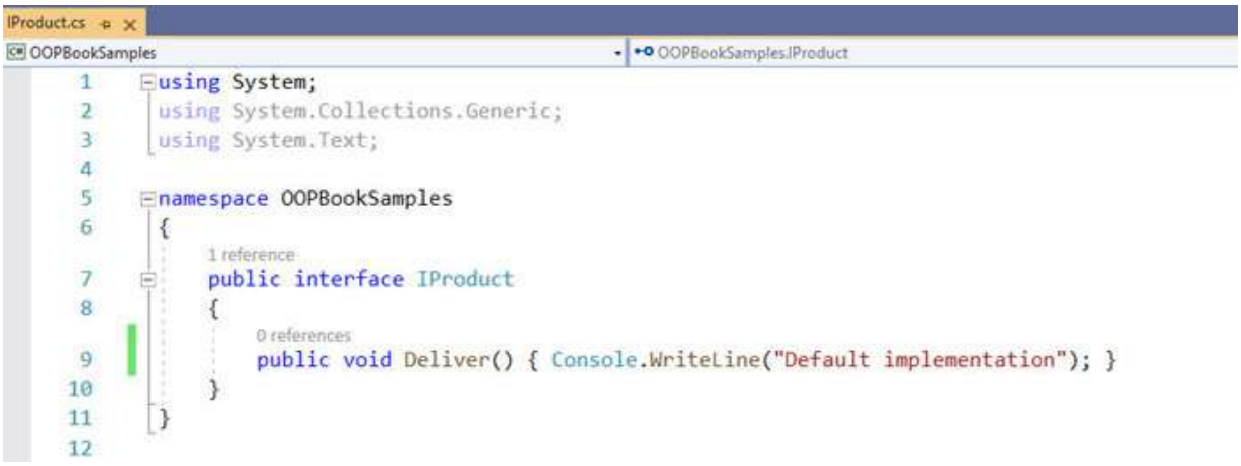


Figure 4.19: *Constructor for struct*

Another aspect of interfaces is that you cannot change or specify access levels regarding properties and methods in interfaces because all of them are public by default. That means using the keywords private, protected, or internal in properties is impossible. The methods and properties can be declared without an access level, as shown in *Figure 4.20*:

Figure 4.20: *Interface without access level*

The use of interfaces can reduce the software's complexity and increase the code's maintainability once they give consistency to the classes and the possibility to apply a pattern across the system. Additionally, the interfaces are essential to make the tests easier at the code level because the implementation would have a more general construction, and a bunch of different tests can be applied to different parts of the system without spending so much energy covered in a test plan.

SOLID Principles

The SOLID principles represent good practices that should be followed by all software developers who build systems using the object-oriented programming paradigm. The principles contribute to having a well-written code, providing maintainability and extensibility. Considering software development is a logical and intellectual process, the same software requirement can be implemented distinctly by different developers, and different approaches can be applied to get the same result.

Good software development is not only about visible software functionalities, but beyond that, it is about having high standards in terms of software architecture and engineering. It is not hard to find a software project that has failed because of technical problems and limitations regarding maintenance and extensibility. In any software development life cycle, the costs of changes get higher when changes are made after production releases. The changes in existent functionalities will likely introduce new issues in the project. In the meantime, new features are being released, thus becoming a great challenge for any software development team to keep the project progressing with high quality.

For that reason, keeping the code clean as much as possible and following the good practices of software development is essential to maintain a long-term solution project that supports relevant changes over time and is ready to absorb always new requirements.

The Single Responsibility Principle is the first of the list defined by the SOLID acronym, and this is frequently applied to software development using object-oriented programming, as it is highly recommended that a class in this paradigm should have a single responsibility. Therefore, the implementation of any part of the software should have a clear and unique objective, being much easier to make changes to that and extend its functionality.

The Open-Closed Principle is the second of the SOLID acronym, and the idea behind this principle is that an object should be open to an extension but closed to changes in its behavior. Any software has a considerable amount of business requirements converted into code, and the requirements can frequently change because of business requests, new regulations, or simply because of a better understanding of the purpose of the software. However, many requirement changes should not represent or require modifications to

the code every time. Suppose a specific routine in the code is being changed several times. In that case, it probably means that the design or pattern used for developing this part of the code is not clear enough and is not following the best software development practices.

Furthermore, each change in the software can cause an impact on production environments by adding errors to an existent functionality, which should be extremely avoided as much as possible. Following the open-closed principle, if a new requirement is requested and it requires changing an existing class or method in the code, the class should be extended to contemplate this new request and not changed. Therefore, adding new requirements should be considered new functionalities and represent extensions instead of modifications.

The Liskov Substitution Principle was defined and created by Barbara Liskov. This good practice's main objective is to avoid throwing exceptions in a system when inheritance is not recommended. Inheritance allows reusing implementation from a parent and common class across the software. Still, when a specific method or property from the parent class does not apply to all the possible children classes, it indicates that the model does not entirely reflect the business requirement. It can generate issues in production environments with unhandled exceptions. Additionally, there is the Interface Segregation Principle, which helps developers to avoid making mistakes using interfaces that do not represent the business requirements. Similar to the Liskov principle, where there is a recommendation of using only the properties and method in a parent class that applies to all child classes, the interface segregation principle applies the same concept but regarding interfaces.

Finally, the last one of the SOLID principles is the Dependency Inversion Principle has the purpose of enforcing the excellent practice of hiding the implementation between classes in cases where there are dependencies between them. To avoid risks in the development process, each class should contain only the implementation related to the class itself and keep low-level loosely coupled in the relation between them

The SOLID principles are the most important concepts of good practices in object-oriented programming and are essential for every developer to know. Understanding and using them allow companies to create stable and readable software and help to reduce the costs of the development process. One of the

premises of any software project is that new requirements will be introduced at any time. Even sophisticated software can have many changes after it is released in the market because any system is flexible and extensible. Good practices that give us confidence in safely making software changes are always welcome, and it is highly recommended to use SOLID principles as an essential part of the coding process.

Conclusion

As seen in this chapter, the object-oriented programming paradigm is the base of C# language. All the logical implementations for business and system requirements are organized in classes, which can be reused, tested, and extended. All the classes must have modifiers that will change the access level of the classes themselves or their properties. Additionally, the C# language contains options to optimize the use of memory (structs) and allows us to give consistency and integrity to the software architecture by applying the concept of interfaces and taking all the benefits of the SOLID principles.

This chapter teaches you how to create classes, properties, and methods and change the access level for each. Additionally, you had the opportunity to familiarize yourself with interfaces, structures, and other types in C# language, allowing you to create and apply the concepts of the object-oriented paradigm in real projects. Finally, you learned the basic concepts of SOLID principles and identify points in a system that can be improved using them.

In the next chapter, you will have the opportunity to learn more complex concepts regarding interfaces and inheritance in object-oriented programming, applying those concepts in more advanced scenarios.

Points to remember

- Objects represent an instance of a class in OOP.
- Interfaces are the contracts that become mandatory for the implementation of methods and properties for any class associated with them.
- SOLID principles are a list of good practices that any developer should use.

Multiple Choice Questions

1. **Which type of structure can be used to specify contracts for classes in C#?**
    a. properties
    b. interfaces
    c. inheritance
    d. protected
2. **How many interfaces can be used associated with a single class?**
    a. one
    b. two
    c. five
    d. unlimited
3. **What is the access level that restricts access to the assembly of the same assembly?**
    a. protected
    b. public
    c. internal
    d. private

Answers

1. **b**
2. **d**
3. **c**

Questions

1. Explain the difference between classes and structs.
2. Explain the meaning of SOLID principles briefly.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5

Interfaces and Inheritance in C#

Introduction

Interfaces and inheritance are essential concepts and features provided by object-oriented programming. Its correct use can bring us the advantage of implementing a robust and extensible software architecture, solving real problems regarding complex implementations, and giving any project an excellent standard to be followed in multiple scenarios.

Learning advanced concepts of interfaces and inheritance gives you a chance to apply those in real-life projects properly and gives you a better understanding of legacy, third-party libraries and mainly the .NET platform, in general since it massively uses interfaces, inheritance, and concepts such as dependency injection and inversion of dependency.

In this chapter, you will have the opportunity to dive deeply into implementing interfaces in C#, using inheritance, understanding how to use its concept in distinct scenarios, implementing testable, reusable functional code, and always taking the benefits of best practices of object-oriented programming.

Structure

In this chapter, we will discuss the following topics:

- Implementation of interfaces
- Testability of interfaces
- Dependency injection concepts
- Concepts of inheritance in C# language

Objectives

After studying this unit, you should be able to understand interface concepts, apply interfaces using C# language, use inheritance in .NET Core projects, create fundamental applications with the best practices of interfaces and inheritances, and build testable classes

Implementation of interfaces

Interfaces in C# are the most efficient way to simulate the behavior of multiple inheritances, considering that .NET does not support multiple inheritances consisting of a single class derived from more than one class. An interface represents a contract that should be followed by any class that implements it; the same happens in legal contracts. Once a method or property is defined in the interface, the class using that interface must implement all the interface specifications as mandatory.

Traditionally, an interface contains just signatures and not implementation. However, C# 8.0 brought a new functionality that allows us to specify default implementation for methods, and it represents an alternative to abstract classes in specific scenarios. By definition, an interface is a group of methods and properties required for each class that needs to use the interface. It gives consistency and integrity to software development and makes it easier to test, change, and extend functionalities.

There are advantages and clear benefits of interfaces because it adds loose coupling to the design and software architecture; the possibility to create reusable components helps in maintainability and decreases the risk of introducing issues in the project once the implementation is separated from the interface, which contains a single specification.

For instance, imagine the need to implement a system that allows users to register different types of products. All the products share similar basic properties and characteristics. Still, they have relevant differences that are reasonable to use an interface containing all the properties common to all products. A few products can have distinct specifications, such as ideal temperature and expiration date. And other products can have even new properties, such as legal requirements and roles based on safety. Considering how various the products could be, it is not possible in object-oriented programming to create a single class that would have all the potential properties and methods for all products and would universally represent every specific situation.

Therefore, *Figure 5.1* represents a simple example of the interface for the product:

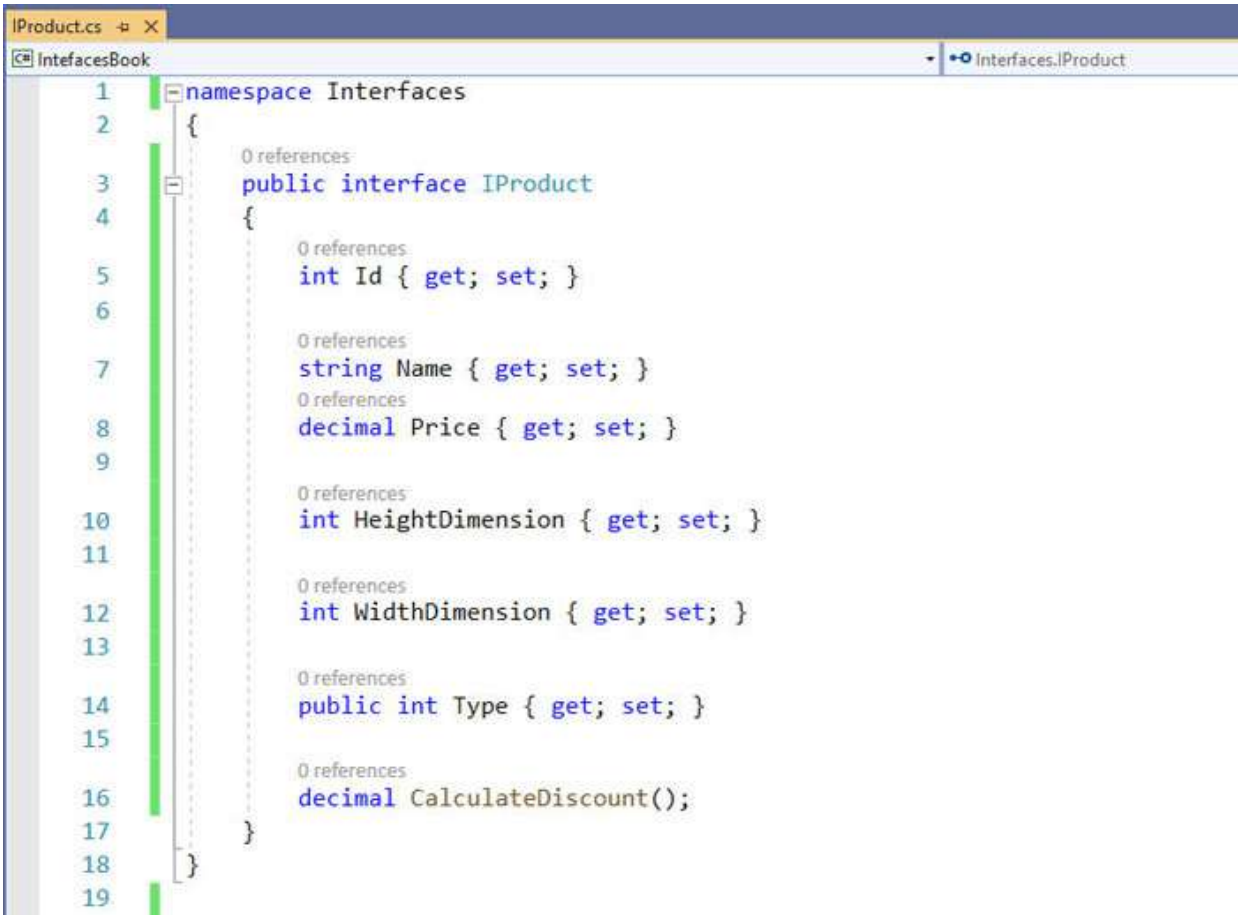


Figure 5.1: *Product interface*

In that case, the interface called IProduct has all the methods and properties that would be shared by all the product classes that will implement its contract. In the given scenario, the classes must define the logical routine for the method responsible for calculating the discount, and the compile will throw an error if the method does not have the same signature and name. The method is a part of the contract and should be followed by every class that refers to that interface. In that way, we guarantee uniformity between the pieces of code that represent the same context. To use interfaces in C# language, you must refer to them after the class name, similar to what is done for inheritance. If the implementation of properties, methods and events is missing in the class, the Visual Studio will show an error indicating which specification of the interface has not been implemented, as shown in *Figure 5.2*:

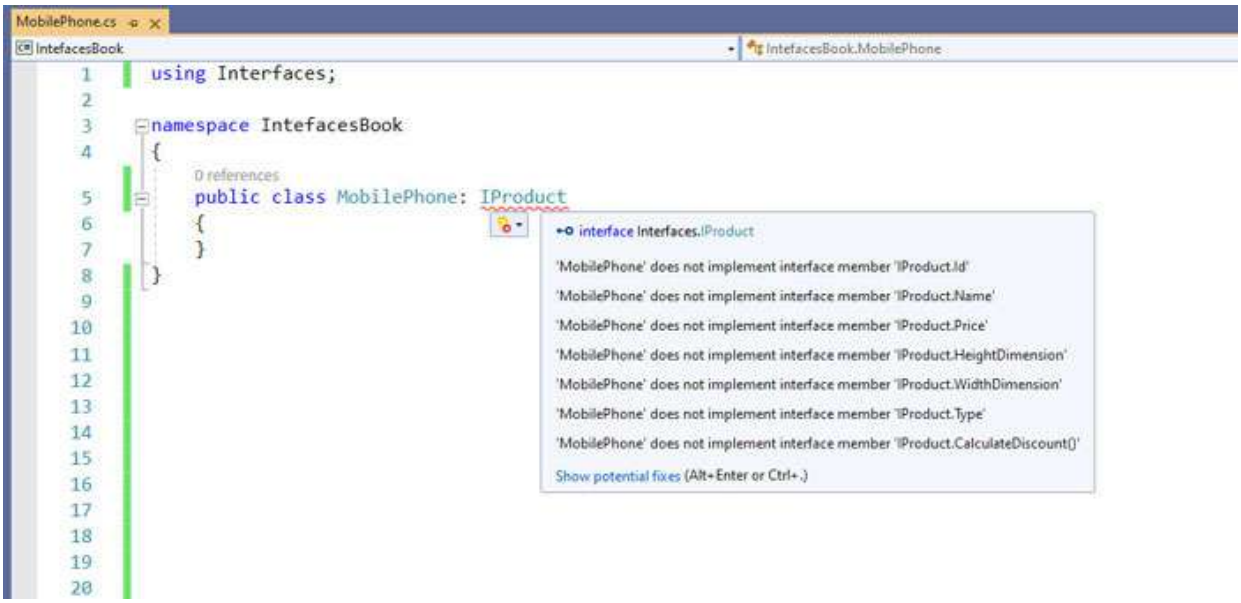


Figure 5.2: *Interface implementation error*

In this example, a class called "**`MobilePhone`**" was created. As the interface was not implemented, it is impossible to compile the project in Visual Studio until the entire interface is implemented. Considering the class can contain properties and methods apart from the required ones defined by the interface, it is possible to give a particular behavior for each product. The correct implementation of the IProduct interface would be as shown in *Figure 5.3*:

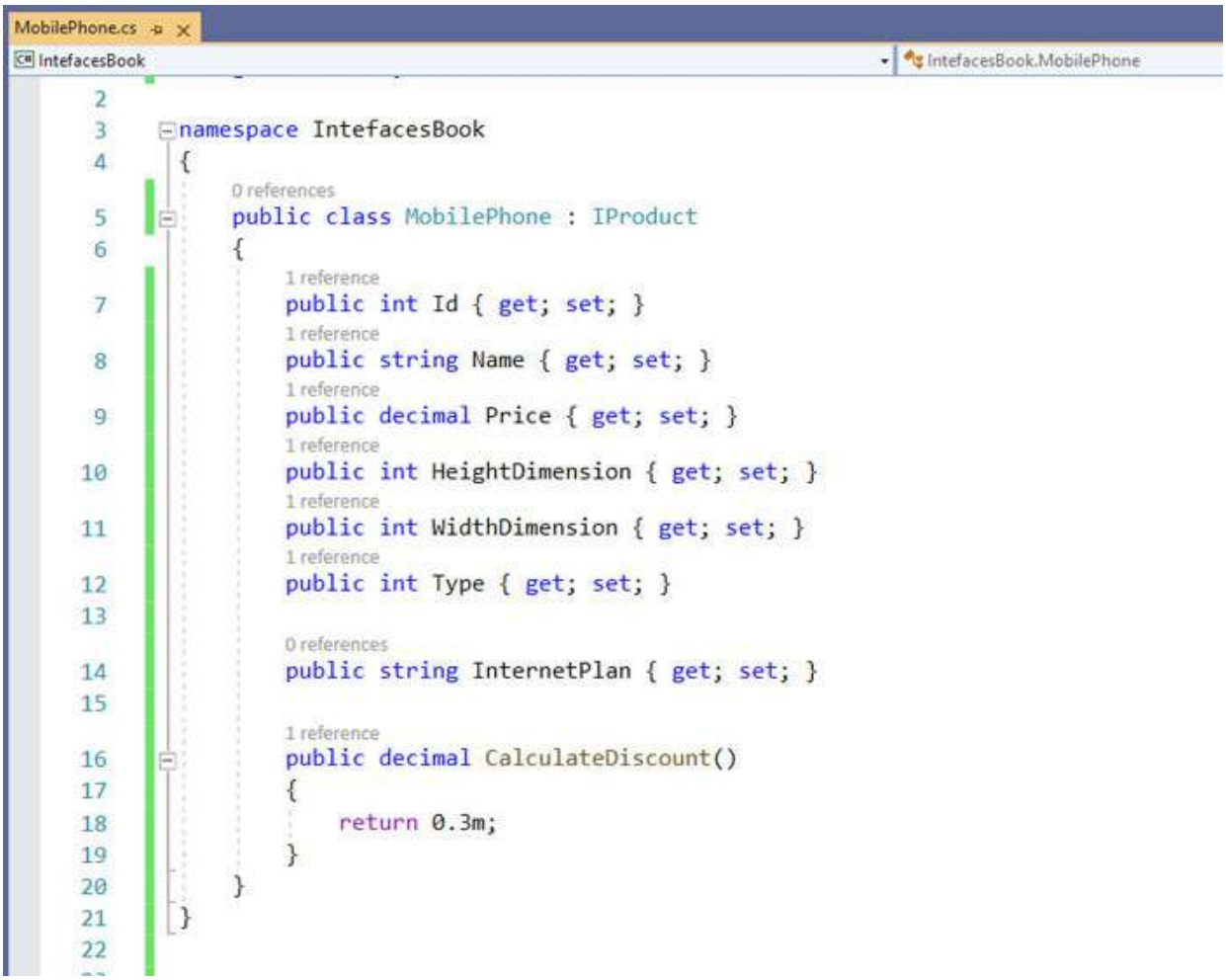


Figure 5.3: *Interface implementation*

After creating the properties and methods with the same signature as required by the interface, Visual Studio does not show any error messages anymore. Further future changes in the interface will directly affect all the classes that contain its implementation. As seen in the implementation class, a property called "**`InternetPlan`**" was created. In that context, it applies only to the Mobile Phone class; therefore, it is possible to extend the classes despite the mandatory implementation defined by the interface. Furthermore, a specific logic for the discount was implemented in the underlying method, which can be particular for each product class.

If most of the products would share similar discounts, as was discussed earlier, it is practical to specify default logical statements in interfaces regarding members and methods, as shown in *Figure 5.4*:

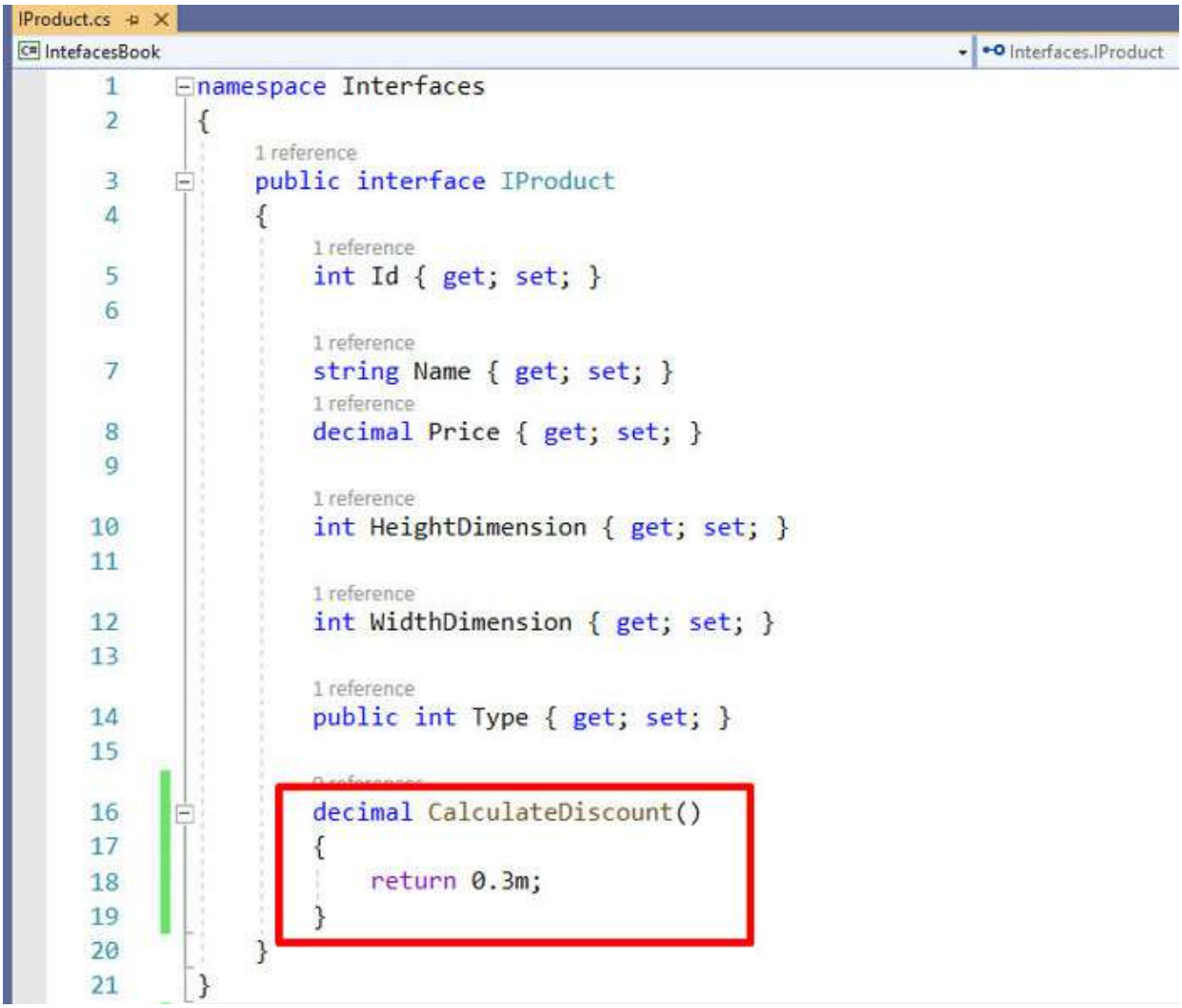


Figure 5.4: *Interface implementation*

In that case, the default value of the discount for all products will be “0.3”, and if the method in the interface has a default implementation, the compiler will not show any errors. That feature is only available since version 8.0 of C#.

In terms of flexibility, the use of interfaces allows us to switch between different classes that implement the same interface, being possible to build generic logical routines that can be easily extended and modified without a significant impact on the software. The interface can be used as a parameter in methods and the creation of variables, as shown in *Figure 5.5*:

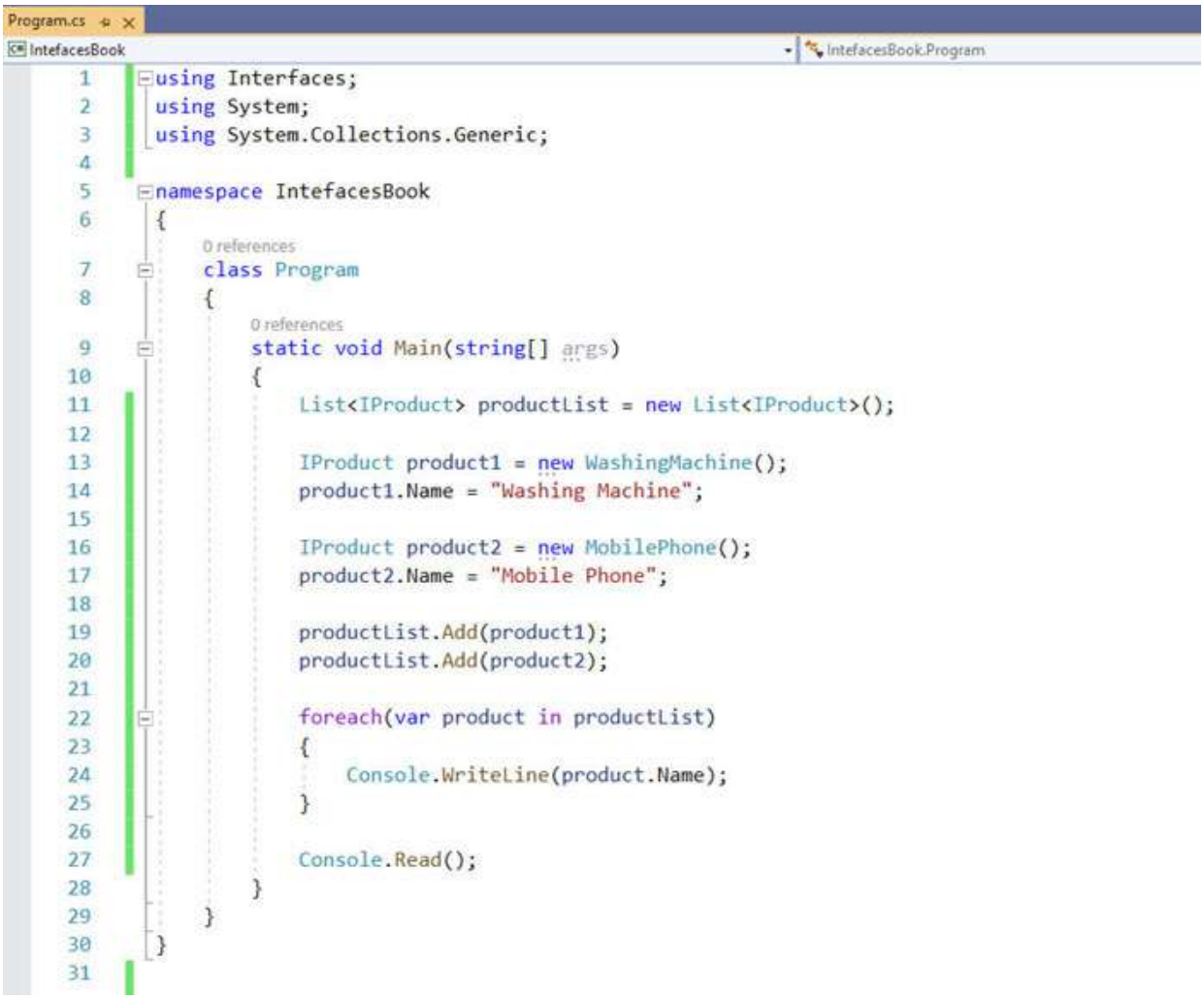


Figure 5.5: *Generic objects with interfaces*

A generic list of IProduct types was created, and the list was populated with two items of different classes: “WashingMachine” and “MobilePhone.” Both of them represent a distinct implementation of IProduct class, but for the compile, they are as they were of the same type because they share the same interface. It allows us to easily extend that functionality by referring more classes for other kinds and making joint operations as if they were the same type, such as looping the list to show results or saving a collection of objects in a database.

Multiple interfaces

Since the first version of the .NET platform, multiple inheritances are not possible for C# language, but a similar result can be achieved using multiple

interfaces, although the limitation of this feature is in comparison to a whole multiple inheritance concept. In our sample scenario, an online store can have products sold only as online products, such as online books and other info products. For these types of products, the system may have specific characteristics that are valid only for this product category. Other properties for non-online products are not suitable for online ones, such as the properties regarding product dimension. Pondering this issue, splitting the interfaces into specialized ones is recommended and that will contain the properties and methods related to the particular categories of products, as shown in *Figure 5.6*:

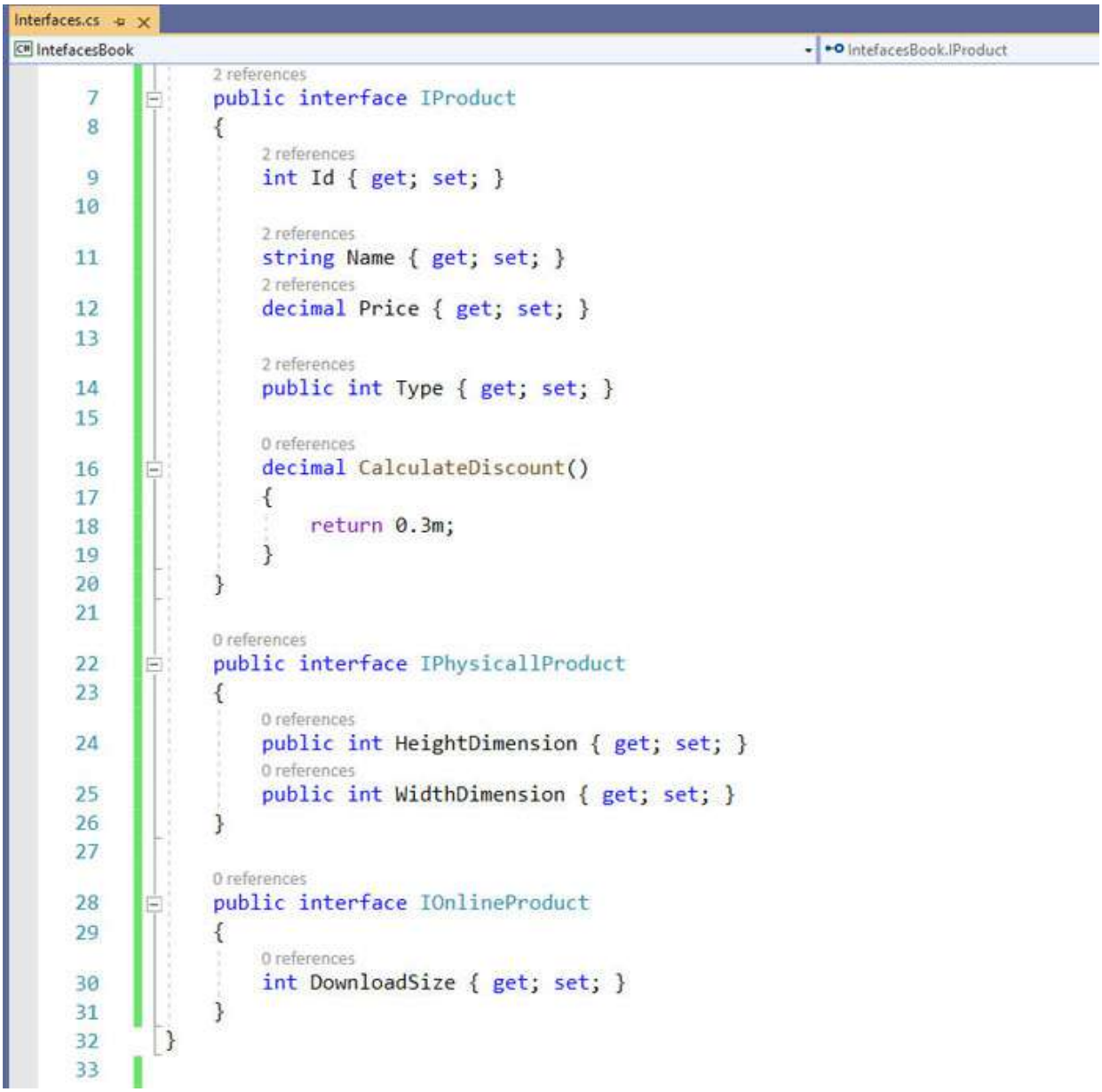


Figure 5.6: *Multiple interfaces*

Instead of having a single generic interface for products, the system might have multiple interfaces that contain specific properties that apply to particular scenarios. For instance, the info related to download size exists just for online products and dimensions only for physical products. All the standard specifications for the available products can belong to the IProduct interface with fewer properties than the previous example. Using the multiple interfaces concept, the Washing Machine class would refer to two interfaces, as shown in *Figure 5.7*:

```
WashingMachine.cs
IntefacesBook                                   IntefacesBook.WashingMachine
1   namespace IntefacesBook
2   {
        0 references
3       public class WashingMachine : IProduct, IPhysicallProduct
4       {
            2 references
5           public int Id { get; set; }
            2 references
6           public string Name { get; set; }
            2 references
7           public decimal Price { get; set; }
            1 reference
8           public int HeightDimension { get; set; }
            1 reference
9           public int WidthDimension { get; set; }
            2 references
10          public int Type { get; set; }
11      }
12  }
13
```

Figure 5.7: *Multiple interfaces*

The properties ID, Name, Price, and Type are being implemented forced by the interface IProduct and the properties **`HeightDimension`** and **`WidthDimension`** are specifications of the **`IPhysicalProduct`**. If the system has an online product such as online books or electronic products, the IOnlineProduct interface should be referred to in the underlying class, as shown in *Figure 5.8*:

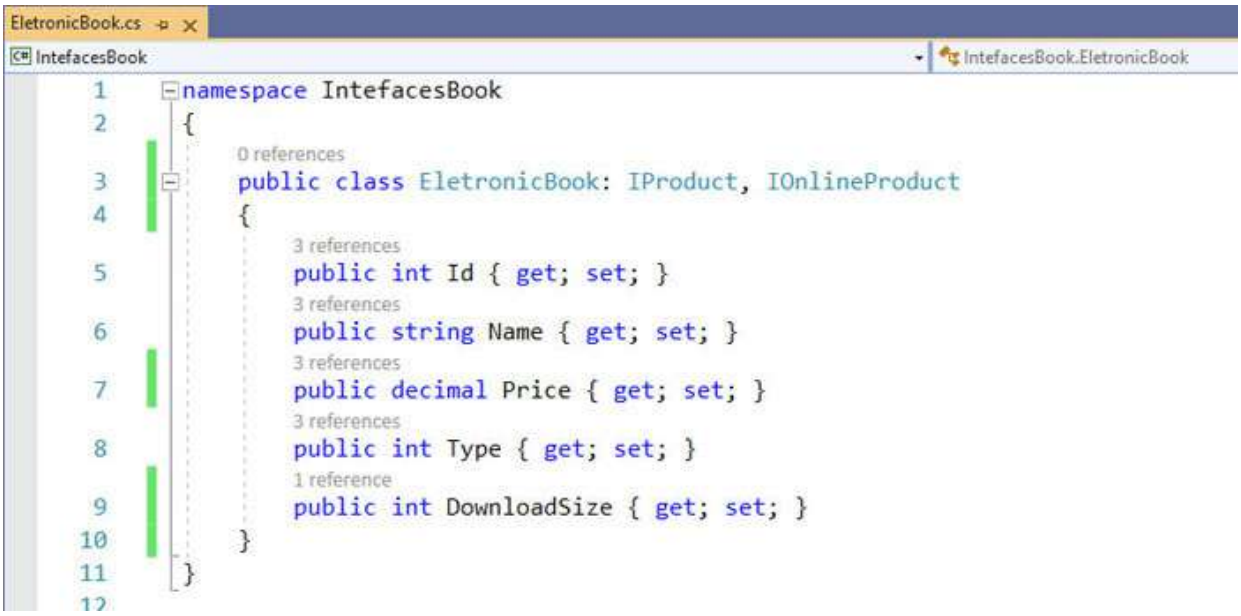


Figure 5.8: *Multiple interfaces*

In that case, the class does not contain any specific reference to dimension, but it implements the property regarding download size. There is no limitation to the number of interfaces used in a class. It gives us flexibility and consistency across the software project in terms of the same pattern being used by all the classes related to products.

Testability of interfaces

One of the significant issues in the market associated with software projects is the difficulty of testing the functionalities efficiently. The leading cause is the high coupling between methods and components in the software, making the software harder to test. There are many ways to reduce the dependency between classes in object-oriented programming, such as SOLID principles. Following the best practices of software development allows us to develop systems where the architecture has high flexibility and maintainability to increase the quality of software.

With the use of interfaces, it is possible to cover more cases of tests faster and easily mock classes that will simulate the original classes we are trying to test. For example, a real case scenario can have a method in a class responsible for making database operations. Regarding the classes that implement the IProduct interfaces, even the software can have hundreds of classes that implement the interface; it is possible to cover an appropriate

amount of code in the tests if the system contains methods that have the IProduct interface as a parameter, instead of the implementation of it, as shown in *Figure 5.9*:

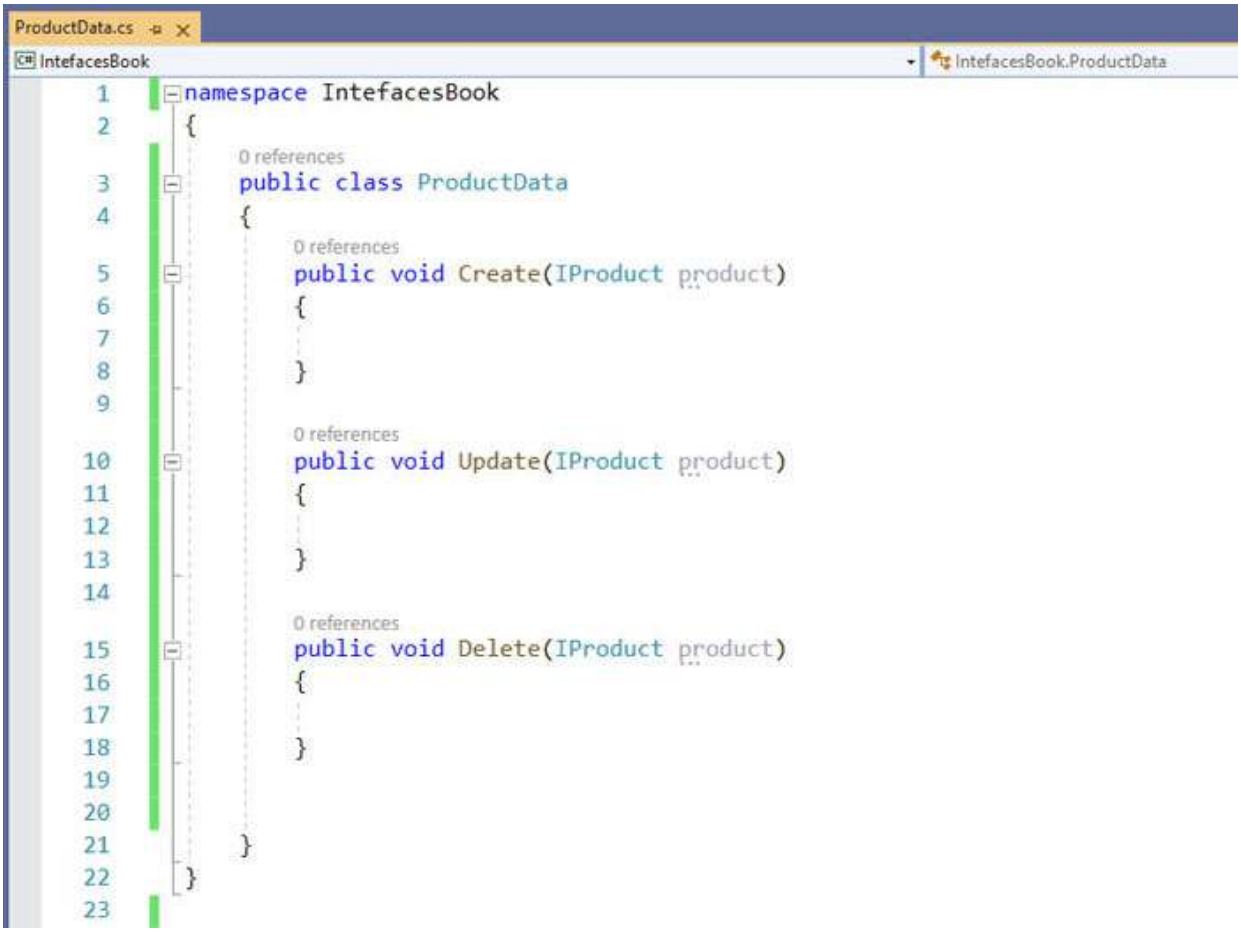


Figure 5.9: *Database operations for products*

As the system can have a wide variety of product types, all of them share the same database operations in the given scenario. It means that the create, update, and delete methods on the class "**`ProductData`**" can be called passing as a parameter for any class that implements the IProduct interface. This approach facilitates the effort in writing unit tests. Also, it helps in the maintainability of software in all the places where the methods are being called, as shown in *Figure 5.10*:

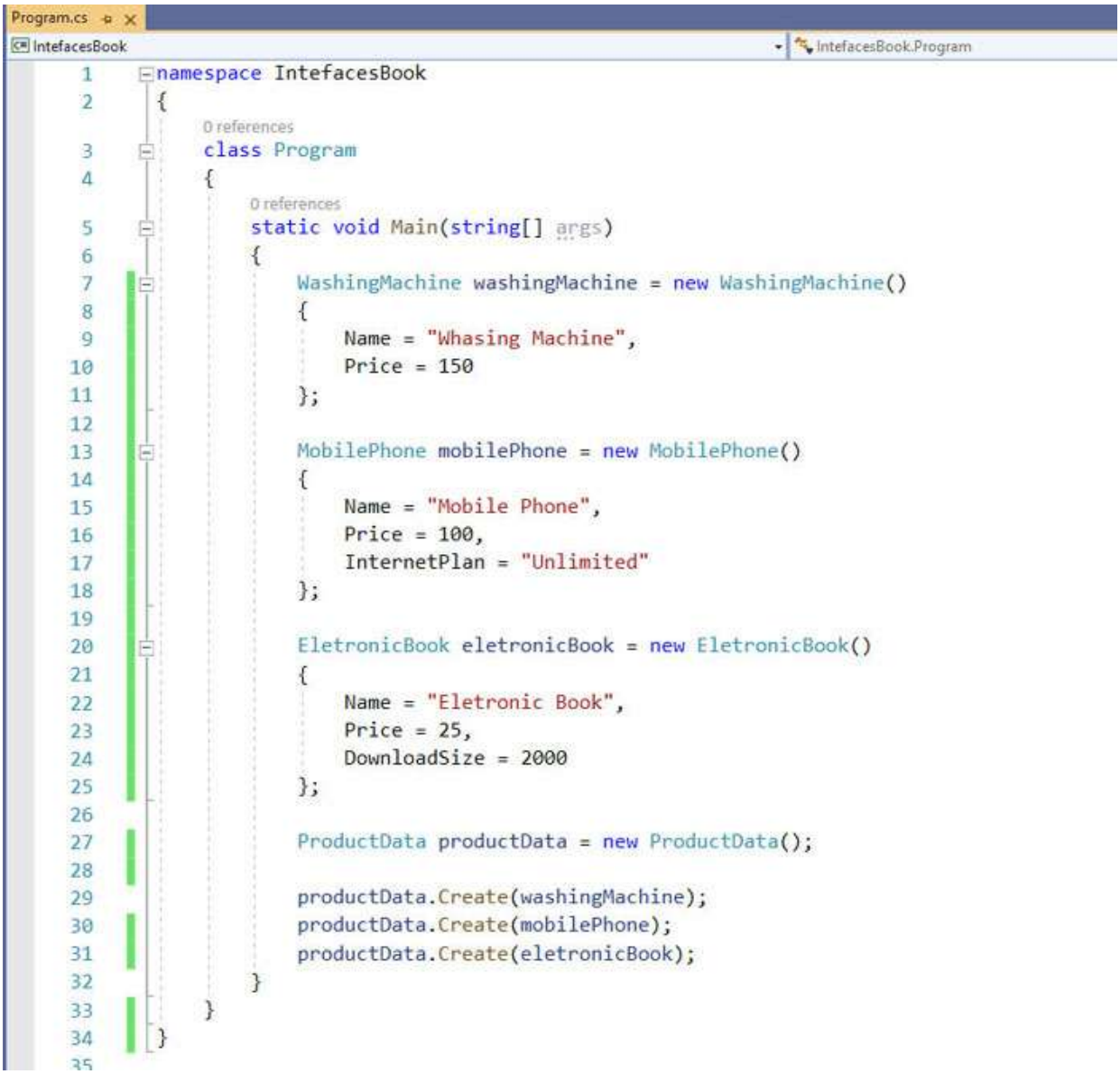


Figure 5.10: *Testable classes using interfaces*

Although the example contains instances of objects from three distinct classes, all of them can be passed as a parameter to the "**`Create`**" method of the class "**`ProductData`**" because the method expects to receive an implementation of the interface IProduct.

Dependency injection

In general, one of the most notable and visible characteristics of .NET Core is the extensive use of the dependency injection concept, mainly in Asp.Net Core applications. Over time many concepts of programming, object-

oriented paradigm, design patterns, and software architecture have become widespread in the market, and they are transformed into common sense as many companies and developers have attested the benefits of certain practices in software development, and dependency injection is one of those. To understand its concept properly, let's take a look at the following *Figure 5.11*:

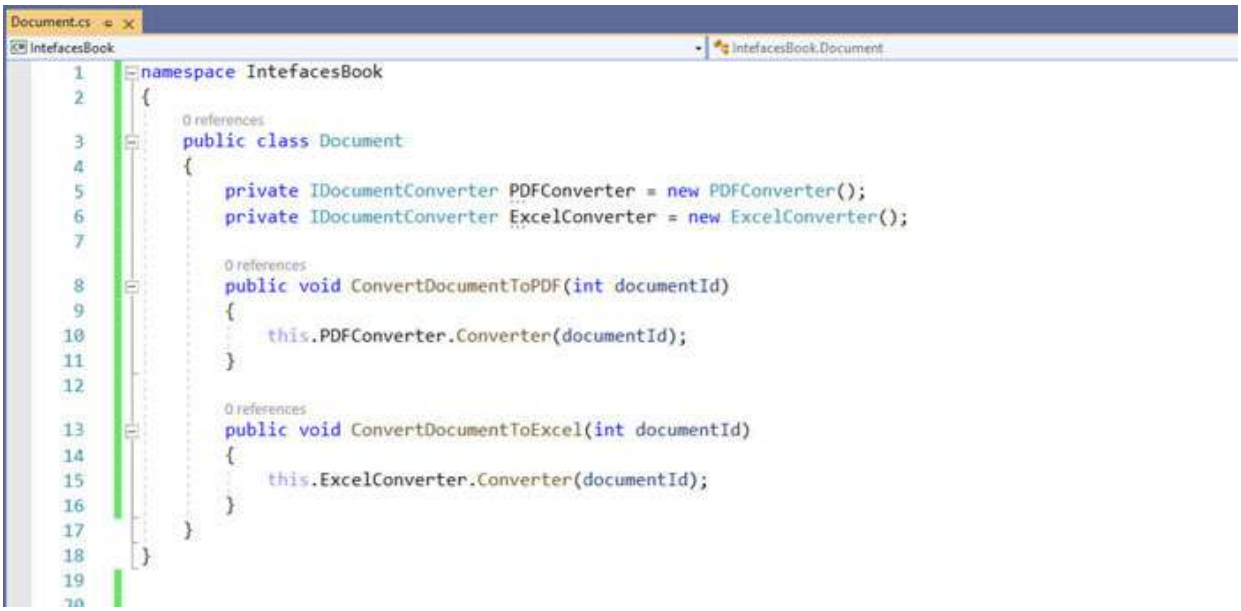


Figure 5.11: *Dependency inversion example*

In this example, there are two explicit dependencies:

- The "Document" class depends on the "PDFConverter" and "ExcelConverter" to fully achieve its objective.
- Suppose the "Document" would have more methods that don't necessarily call any method from external classes. The "Document" class can work without the other two associated classes, but the strong dependency is still there.

In other words, the document class directly depends on the PDFCoverter and ExcelConverter classes, and obviously, they represent dependencies of the Document class. Considering that usually, the conversion of the document involves third-party libraries, depending on how we implement the Document class, its maintainability and testability will be harmed. The associated classes in the example are highly coupled to the document class because this implementation violates the open-close principle, which states

that the classes in object-oriented programming should be open for extensions but closed for modifications in their behavior.

There are many good technical ways to reduce the dependency between classes. In the example, a better design can be achieved by passing the parameters in the constructor of the Document class as interfaces instead of creating instances of it as a private at the beginning of the class, as demonstrated in *Figure 5.12*:

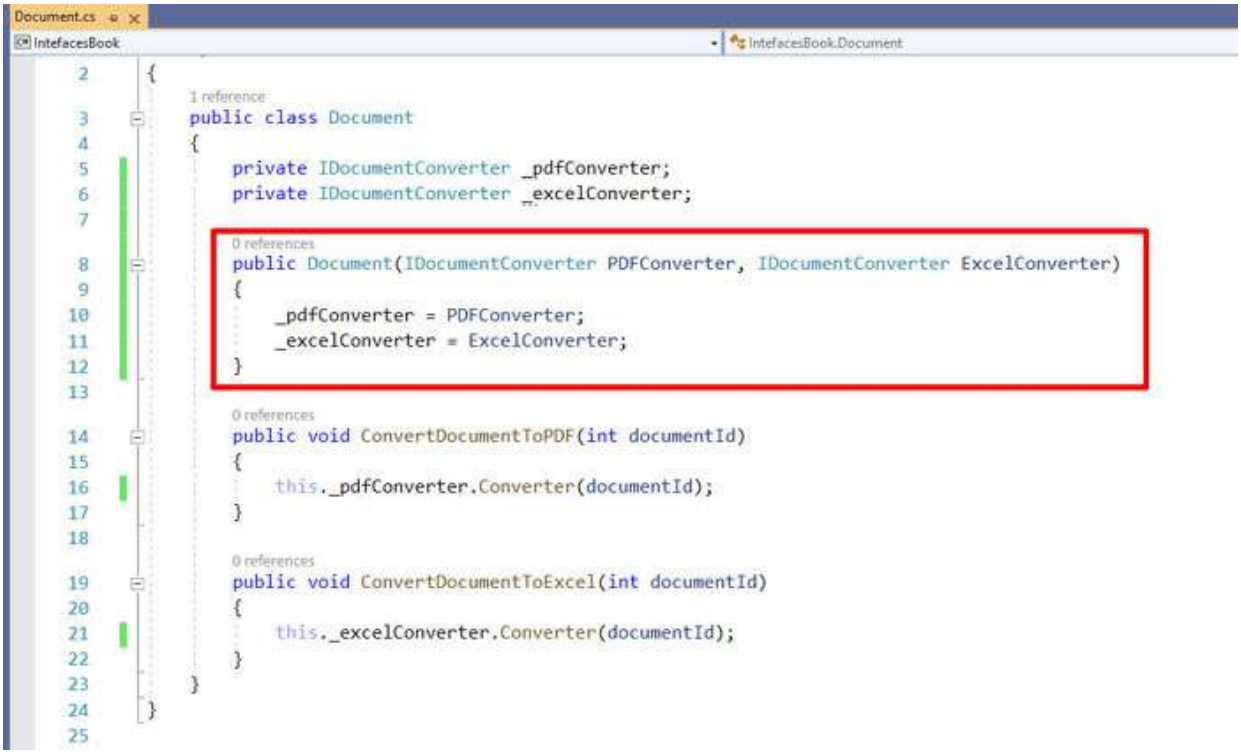


Figure 5.12: *Document class using dependency injection*

With this approach, you can implement PDFConverter and ExcelConverter classes in the Document class and reduce the coupling between all these classes. Beyond that, it consistently improves the testability and reusability of the Document class once it is possible to mock the conversion classes in case it is necessary to apply tests in the Document class without needing a dependency on the third-party libraries. Another important extra factor is that if there is a software requirement change that includes the support of conversion to other formats, the modification in the Document class would be minimal if the new converter class implements the same IDocumentConverter interface.

Inheritance in C# language

Inheritance is one of the most basic features in object-oriented programming, once reusability is a central point in this paradigm. This feature allows us to create child classes that inherit methods, properties, and events of a parent class, being possible to overwrite and specialize the behavior in the child class. As mentioned earlier in this book, the C# language allows us to use only one inheritance. However, it is possible to achieve a feature similar to multiple inheritances using inheritance at multiple levels or interfaces.

Considering the example that was used in the previous section regarding interfaces, the use of inheritance for the product class would have a parent class for the Product, as shown in *Figure 5.13*:

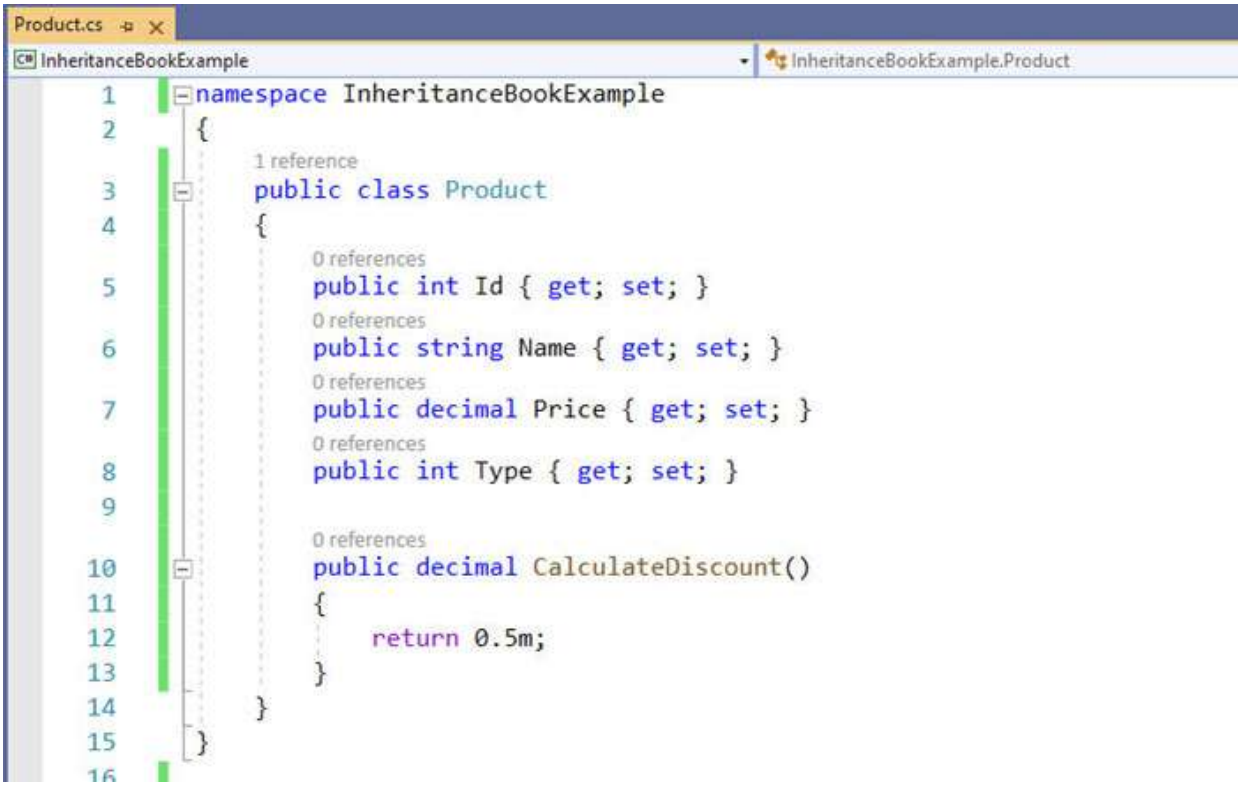


Figure 5.13: *Product base class*

With this base class, any child class must refer to the Product class, being possible to specify other properties and methods, as shown in *Figure 5.14* regarding the MobilePhone class implementation:

```
MobilePhone.cs
InheritanceBookExample                          InheritanceBookExample.MobilePhone
1   namespace InheritanceBookExample
2   {
        0 references
3       public class MobilePhone : Product
4       {
5
            0 references
6           public int HeightDimension { get; set; }
            0 references
7           public int WidthDimension { get; set; }
            0 references
8           public string InternetPlan { get; set; }
9
10      }
11  }
12
```

Figure 5.14: *Mobile phone class as a child*

Considering the class inherits of Product class, it is unnecessary to specify any properties and methods in the parent class. The MobilePhone class can only have specific properties and methods that apply to the child class. If we try to use any members from the parent class, then the compile for C# language will interpret as they would belong to the child class, as shown in *Figure 5.15*:

```
Program.cs
InheritanceBookExample                          InheritanceBookExample.Program
1   namespace InheritanceBookExample
2   {
        0 references
3       class Program
4       {
            0 references
5           static void Main(string[] args)
6           {
7               MobilePhone mobilePhone = new MobilePhone();
8               mobilePhone.Name = "Mobile Phone";
9               mobilePhone.Price = 100;
10              mobilePhone.InternetPlan = "Unlimited";
11          }
12      }
13  }
14
```

Figure 5.15: *The use of mobile phone class*

The properties Name and Price are visible after creating an instance of MobilePhone class as they would be part of it. Considering a system in that context could have hundreds of distinct types of products, that is one of the advantages of using inheritance because all the implementation from a single parent class is being reused, and effort to implement the software is significantly reduced.

The inheritance contains aspects regarding accessibility that can be used to determine the visibility of any of the members from the parent class. If a property or method is specified as private, it is visible only by the class itself and not by the child class, as shown in *Figure 5.16*:

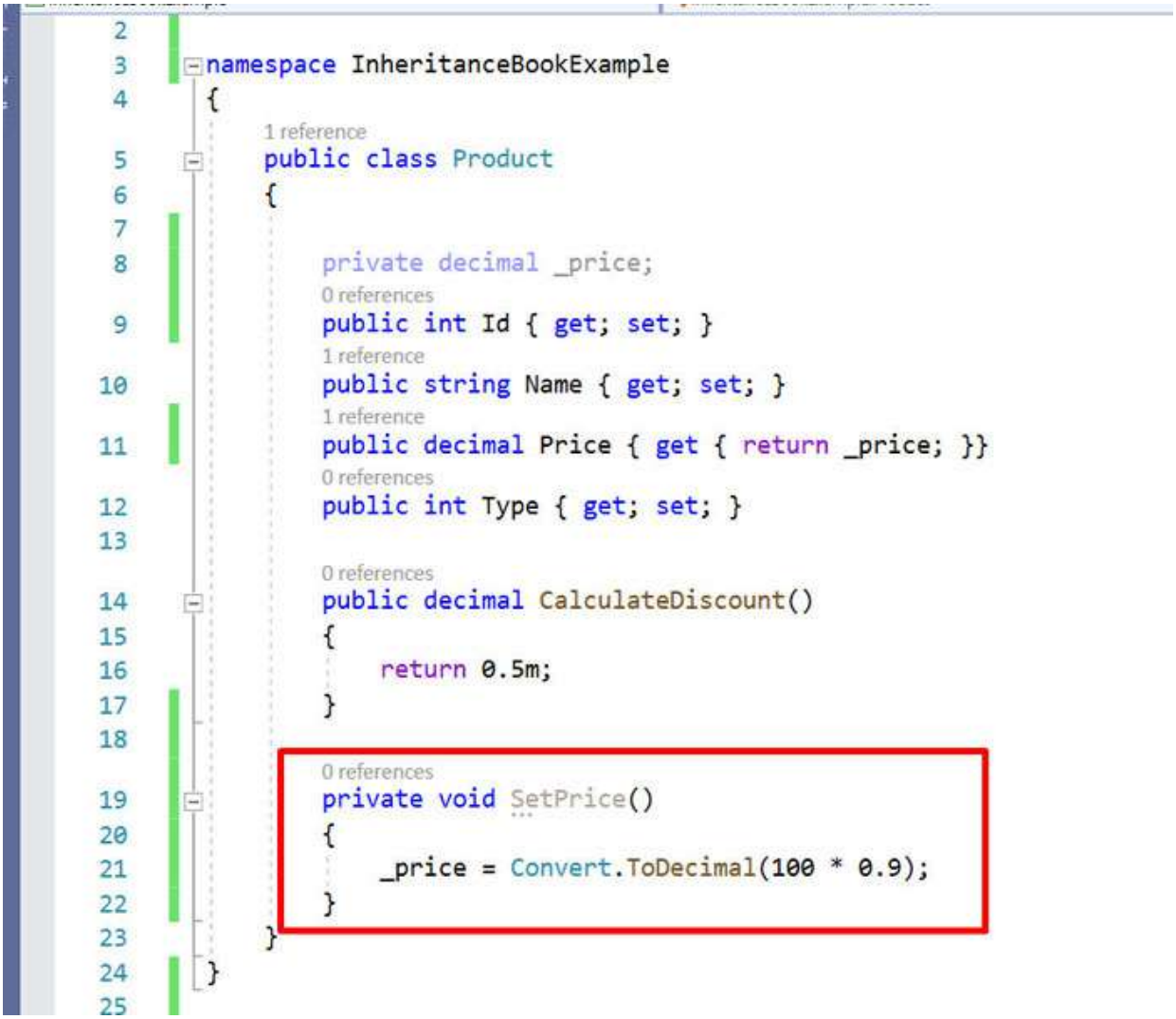


Figure 5.16: *Private method*

Considering the use of the visibility modifier for the method **`SetPrice`**, it is only accessible by the Product class and not for any child that inherits from the class. The same concept applies to properties and events. There are three different ways to provide the accessibility for the SetPrice method to derived classes:

- The method can be changed to the public, which means that any other class, derived or not, can access the method. In the Product class example, it is being applied for the properties Id, Name, and Price.
- It is possible to change the modifier to internal. In that case, the method will be visible by any other class that is in the same assembly as the base class. This feature is quite helpful in cases where there are requirements to avoid the access of specific methods or classes between layers in the systems. For instance, we may want to avoid a method of the Database layer being called and used directly in the View layer. To avoid this situation, the database and view layer should belong to separate projects, and the methods in the database layer should be marked as internal.
- If it is necessary to restrict the visibility of a method to derived classes, the protected modifier must be used to achieve this objective.

A critical feature in inheritance is the ability to override members if the derived class needs to change the default behavior provided by the parent class. Applying inheritance with the possibility of changing the behavior of methods by child classes is compatible with the idea of polymorphism, as the classes relate to each other, but they may perform different tasks when executing methods.

In the example, if the keyword "**`virtual`**" is included in the SetPrice method signature, all the derived classes are allowed to implement and specialize the behavior of the method using the override keyword, as shown in *Figure 5.17*:

Figure 5.17: *Override in the derived class*

Implementing the method in the "MobilePhone" class is slightly different from the parent class. Considering the method in the Product class is marked as virtual, a custom implementation in any child class is optional.

The inheritance in object-oriented programming allows us to extensively reuse code across the system, reduce the time spent writing code, and maintain consistency in implementing business requirements, as the classes in a project must represent real-world scenarios.

Conclusion

Interfaces and inheritance are essential features in C# language that gives us flexibility and bring to our projects high standards in terms of extensibility, maintainability, testability, and other extra benefits, which are common sense in the market associated with software architecture and good practices of software development, reducing the costs of implementation, the effort spent

in building new projects and significantly minimize the risks reference to legacy projects.

In this chapter, you learned how to create interfaces, apply its concept in different scenarios, and how to take the benefits of building testable, reusable, and extensible applications using the C# language. Also, you had the opportunity to learn about inheritance in an object-oriented paradigm and got familiar with the accessibility and visibility of class members.

In the next chapter, you will have the chance to learn the basic concepts of design patterns and how to apply them in different scenarios to build high-standard enterprise applications and solve problems that you could face in many projects.

Points to Remember

- The C# language does not support multiple inheritances but can use multiple interfaces.
- Interfaces give consistency to the software architecture and increase aspects of the testability of the classes.
- Dependency injection is primarily used in .NET projects, and its concept helps us to reduce the dependency between classes and components.
- Inheritance is the best way to reuse code in the OOP paradigm.

Multiple Choice Questions

1. **Which version of C# introduced the possibility of having default implementation in interfaces?**

    a. 7.2

    b. 8.0

    c. 1.0

    d. C# does not support this feature

2. **How many inheritances are allowed in the C# language associated with a class?**

    a. one

b. two
c. five
d. unlimited

3. **What modifier restricts access to a member of the same class?**

a. protected
b. public
c. internal
d. private

4. **How many interfaces can a class implement?**

a. two
b. unlimited
c. none
d. one

Answers

1. **b**
2. **a**
3. **d**
4. **b**

Questions

1. Explain the difference between parent class and interface.
2. Explain the concept of dependency injection.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Basic Concepts of Design Patterns

Introduction

The object-oriented programming paradigm is a fundamental knowledge in software development, which allows us to technically implement complex scenarios that represent a massive challenge in keeping high quality and following the best coding practices. To help developers create extensible and flexible enterprise applications, a series of patterns in terms of software architecture was stated in the market and focused on solving problems in any software project.

Learning the basic concepts of design patterns gives you a chance to apply those in real-life projects properly and provides a better understanding of projects based on the .NET Core platform. Nowadays, the knowledge of the principal design patterns in C# language is essential to achieve the full potential of the crucial benefits of object-oriented programming.

In this chapter, you will have the opportunity to overview the most common design patterns using the C# language and understand how to use their concepts in real projects, implement applications in distinct scenarios, and meet appropriate situations.

Structure

In this chapter, we will discuss the following topics:

- General concepts of design patterns
- Singleton, Façade, and Adapter patterns
- Observer, Builder, and Factory patterns

Objectives

After studying this unit, you should be able to understand design pattern concepts.

You will be able to apply design patterns using C# language and also understand in which scenarios the design patterns can be used

General concepts of design patterns

If we compare its maturity to other more consolidated sciences, such as civil engineering and mathematics, software engineering is a relatively new area of knowledge. A software project has an abstract structure and high mutability, characteristics that make the software development process equally abstract and complex. Once civil engineers do a traditional building project, it is possible to build prototypes, make simulations, and calculate to ensure that the location and planned resources are enough to finish the project on time and meet the expected results. In this process, other similar buildings are usually verified to get all the inputs learned and documented from past experiences to follow the good practices of the market and avoid making similar mistakes that were made in other related projects.

Considering civil engineering is a millennial science, the documentation and knowledge gained from many projects are vast and relevant, and all the practices are well shared among all the professionals in the related area. Software engineering can be compared with other areas once software projects, in general, can share similar problems that need to be solved in terms of architecture and implementation. Therefore, it is possible to determine a guideline for good practices in software development and models that can be used in projects with similar structures and requirements.

Considering that context, Design Patterns in software development represent a collection of solutions for problems that frequently can be found in real scenarios, and in general, they are stated by the technical community and professionals in the market that have spent the necessary time identifying these patterns and verifying the level of their occurrence in distinct projects. In the following sections of this chapter, you will have the opportunity to familiarize yourself with the following patterns, which are common in .NET Core applications: Singleton, Façade, Adapter, Observer, Builder, Factory, Model-View-Controller, and Decorator.

Singleton pattern

The Singleton pattern is one of the most used and known design patterns among .NET developers once its objective matches many scenarios used in web and desktop applications. As its name suggests, the Singleton pattern consists of keeping a single instance of a specific object for the entire application; the single object is globally accessible. This kind of implementation can be helpful in terms of performance and consumption of resources in the server once a single instance of the object is presented in the memory. A shared instance of an object is primarily used for caching operations, logs, database connections, and HTTP requests. There are some advantages to this approach, depending on the scenario. In other scenarios, using a singleton pattern is not recommended, mainly in cases where the object's state must be different across the system in multiple concurrent accesses. For example, imagine a scenario where it is necessary to implement a routine in an online store project to record the transaction logs in an external resource like a text file.

Considering there is a limitation of concurrence; in that case, it is possible to write only one log at a time, and the system needs to be designed to support that requirement and guarantee that the entire application will use a single instance of the object responsible for writing these log operations. The proposed scenario is represented in the following image:

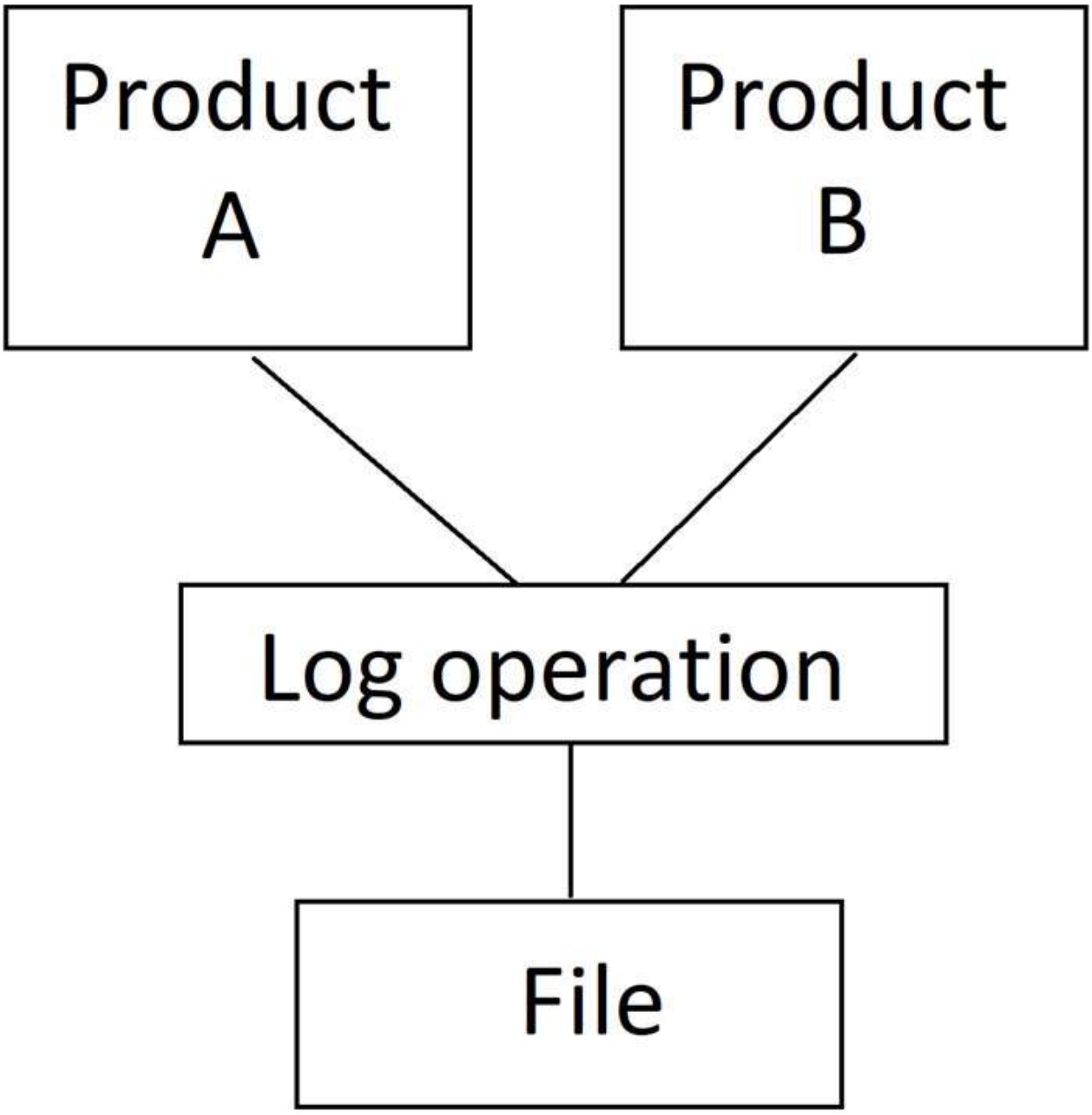


Figure 6.1: *Log operation*

Considering the access to the object responsible for making the log operations can happen simultaneously in the system, if the code is not designed to prevent concurrence, there will be a huge chance to have multiple attempts to write in the file at the same time, which will cause an error in the system; considering the operating system will block the file while a log is being recorded. The singleton pattern is an excellent alternative to avoid unexpected multiple simultaneous access to the file. To achieve this objective, it is necessary to use the static class concept in C#

language and guarantee that access to the object is locked when the object is currently being used in a call, as shown in *Figure 6.2*:

Figure 6.2: *Log operation*

In this example, there is a class called "**`OperationLogger`**" with a static property that contains a verification if a single instance of the logger is null. The routine guarantees that a new instance is created inside a lock statement if null. Every place and all the time this class is called, verification in the lock property will be made, and access to the resource will be locked. That means if there are many accesses to the static property in the system, they can only get it one after the other sequentially. The lock statement will force the subsequent request to wait to be processed. It allows the system to write the log once at a time and avoid concurrence in the resource.

The Asp.Net Core Web applications give the possibility of using singleton pattern combined with dependency injection concept in the initialization of the application, as shown in *Figure 6.3*:

Figure 6.3: *Singleton in Asp.Net Core applications*

In that case, the class itself does not need to have a static property implementing the singleton pattern because the .NET Core application will guarantee the singleton pattern behavior for the class across the application. Every place where an object of this class will be injected will use the same instance. The singleton pattern is massively used in .NET Core, mainly for external resource interaction such as HTTP requests, database connections, and other purposes.

Façade pattern

The primary purpose of the Façade pattern is to simplify the use of complex operations between systems, modules, or projects in a solution, allowing abstracting multiple related operations in a higher-level interface to be easily consumed. One of the good practices of software architecture is the concept of encapsulation, which means that other structures must use a method or class in the software without necessarily providing information on the implementation they are making internally. Therefore, it is recommended not to expose the complexity of classes in object-oriented programming, and the use of specific resources or methods will be transparent to other parts of the software. The Façade pattern helps us to achieve this objective once it is possible to abstract the complexity of various complex and combined operations using a simple and unique interface that will be provided to other

classes across the system. This approach is primarily used in integrations between systems and clearly, simplifies the communication between distinct components; always considering the fact that one side of the integration surely will change in complexity, evolving its functionality or implementing new business requirements.

For example, imagine a scenario where an online store needs to consume products and services from different companies and even other online stores. Even if the products come from different systems and distinct locations, the experience for the user in the frontend layer will abstract that complexity. Even though the source is different for each type of product, there are similar points between all those in terms of transaction and payment. For the final user, the experience would be like a single system has been accessed, and the complexity of the consumption of many subsystems will be transparent, not only for the final users themselves but for the frontend layer in the software as well. Creating a single access point for all these services will help in the maintenance, evolution, and tests of the software once a unique structure is kept centralizing those operations, as shown in *Figure 6.4*:

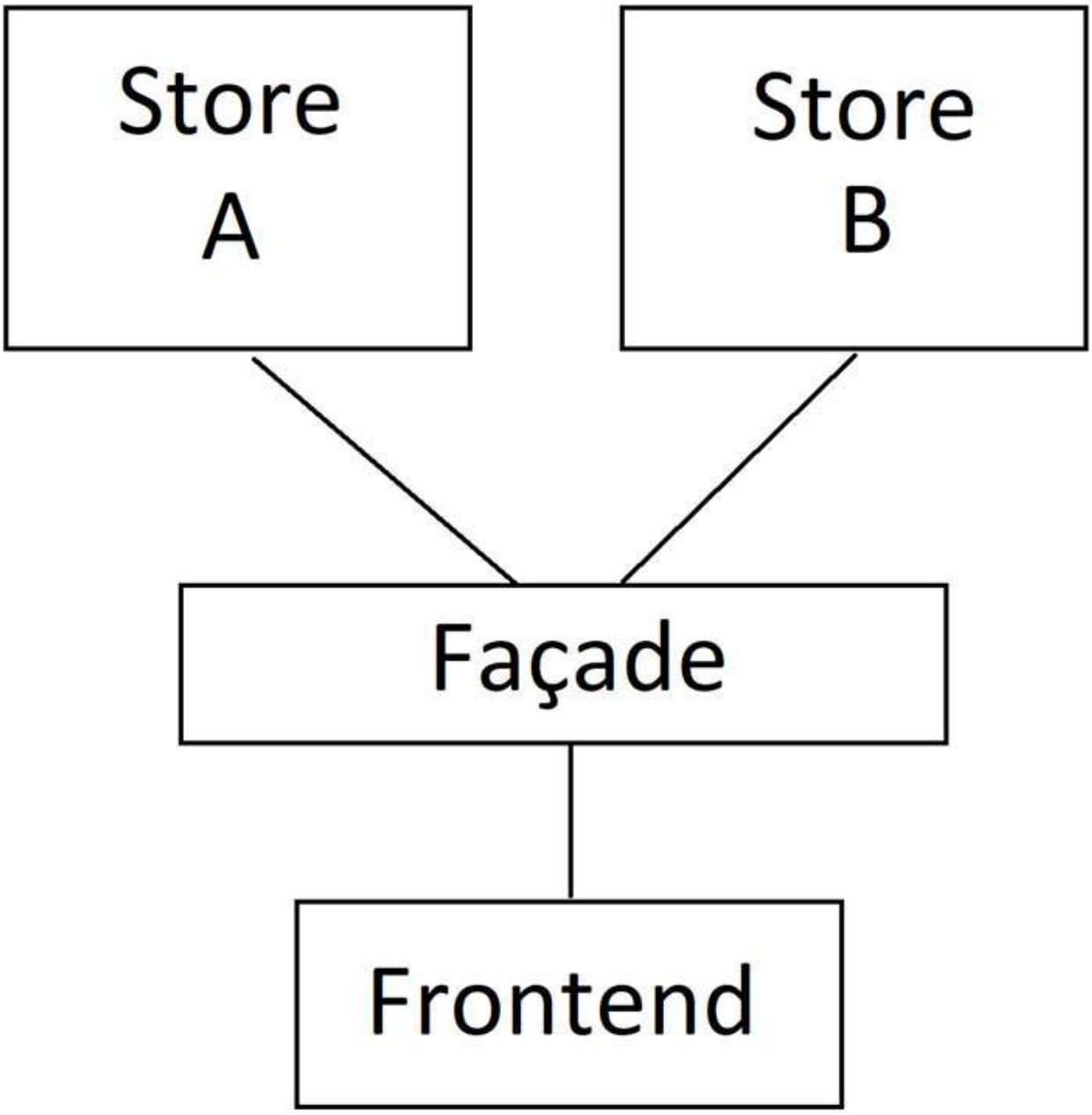
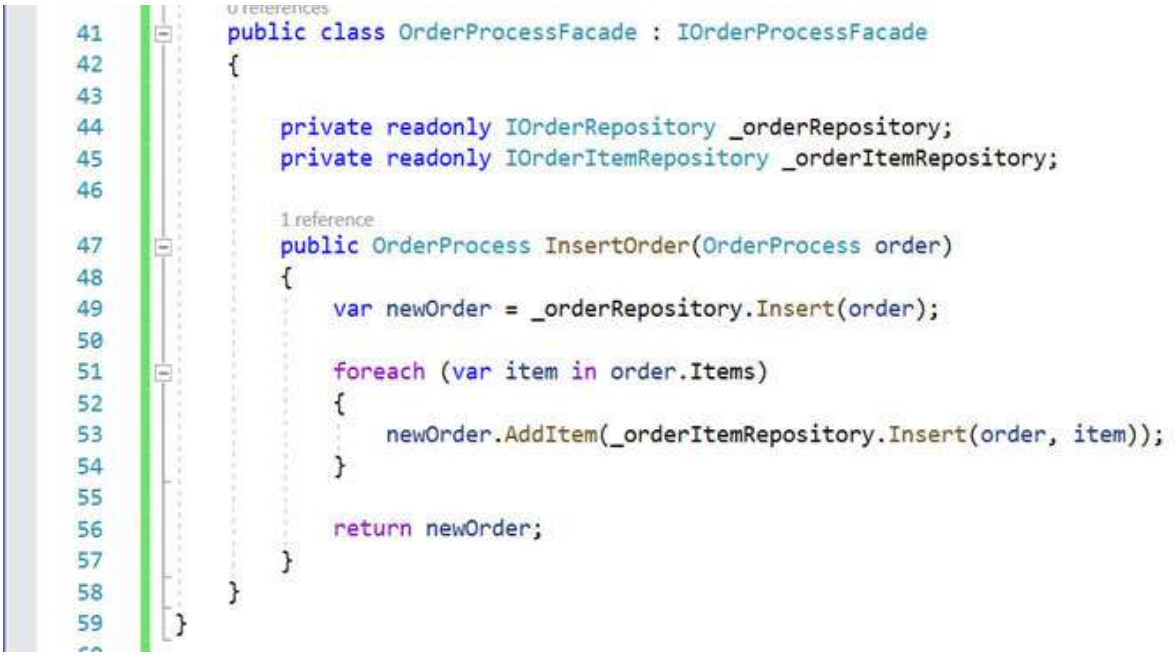


Figure 6.4: *Façade pattern*

Considering a real scenario might have hundreds of distinct types of projects, it is better to create a single point of implementation for handling all the operations regarding all the common operations that could be made by an order process, as shown in *Figure 6.5*:



```
0 references
public class OrderProcessFacade : IOrderProcessFacade
{

    private readonly IOrderRepository _orderRepository;
    private readonly IOrderItemRepository _orderItemRepository;

    1 reference
    public OrderProcess InsertOrder(OrderProcess order)
    {
        var newOrder = _orderRepository.Insert(order);

        foreach (var item in order.Items)
        {
            newOrder.AddItem(_orderItemRepository.Insert(order, item));
        }

        return newOrder;
    }
}
}
```

Figure 6.5: *Façade example*

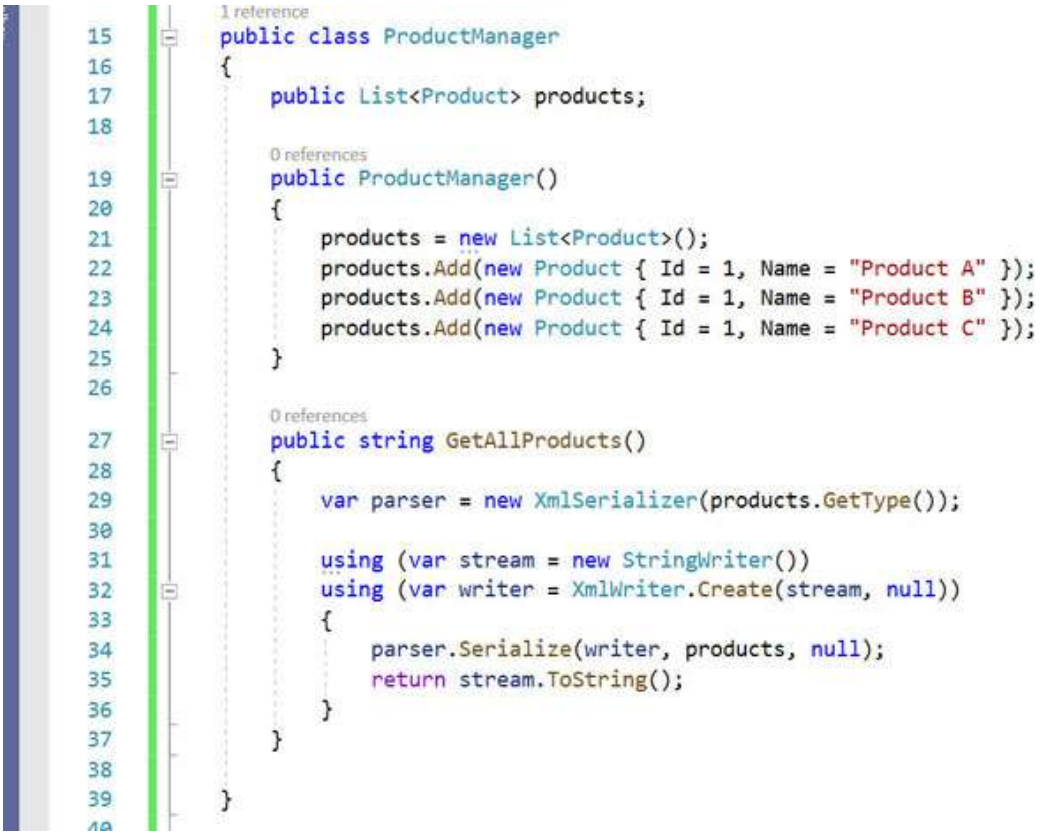
Instead of directly using a class called "**`OrderProcess`**", a centralized class was created to be a single access point to multiple classes that implement the common interfaces **`IOrderRepository`** and **`IOrderItemRepository`**. To keep good coding practices, if a particular pattern is used, we must refer to the pattern name in the interface or class name as was done with the class "**`OrderProcessFacade`**." In software development, many concepts are related to each other. It is possible to easily correlate the Façade pattern with other principles in the object-oriented programming paradigm, such as S.O.L.I.D principles. All design patterns' primary objective is to give the software high-maintainability and flexible architecture.

Adapter pattern

The Adapter pattern is one of the most used patterns in projects of software that implement object-oriented programming considering almost all the libraries responsible for connecting applications with databases using that pattern in the .NET platform to have a simple and common pattern between the C# language and database operations. This pattern is responsible for converting a specific implementation to one understandable by the software, which is consuming that resource, increasing the reusability of libraries in legacy and different projects. This pattern is highly recommended when a

class does not have an interface but at the same time needs to be primarily used. Furthermore, it is possible to implement this pattern when an adapter needs to be created without changing the interface used by an existing class.

For example, imagine a scenario where there is a system that uses a specific class to get product information for an online store. The same class has a method to return a list of products in a specific format (XML), but we need to handle this information across the system in JSON format instead of it. In software development, there are many ways to achieve this objective: one of them could be changing the original class to return the list of products in the expected format or even converting the list of products to JSON wherever needed. But, considering the good practices of software development, which state that avoiding behavior changes in an existing class is recommended, we must extend the functionality to achieve the objective instead of changing the original class. The original class in this hypothetical scenario has a method called "**`GetAllProducts`**," which returns an XML with a list of products, as shown in *Figure 6.6*:



```
1 reference
15  public class ProductManager
16  {
17      public List<Product> products;
18
        0 references
19      public ProductManager()
20      {
21          products = new List<Product>();
22          products.Add(new Product { Id = 1, Name = "Product A" });
23          products.Add(new Product { Id = 1, Name = "Product B" });
24          products.Add(new Product { Id = 1, Name = "Product C" });
25      }
26
        0 references
27      public string GetAllProducts()
28      {
29          var parser = new XmlSerializer(products.GetType());
30
31          using (var stream = new StringWriter())
32          using (var writer = XmlWriter.Create(stream, null))
33          {
34              parser.Serialize(writer, products, null);
35              return stream.ToString();
36          }
37      }
38
39  }
```

Figure 6.6: *Product Manager class*

The class states that XML is the result expected for all the clients that will be consuming this method, which in real scenarios is not necessarily true. One possibility would be to ask the library owner to provide the return in other formats, which might not be a reasonable option when external libraries are being used. An alternative would keep the original class as it is and extend the functionality creating an adapter to convert from XML to JSON in a centralized place, as shown in *Figure 6.7*:

```
43  public interface IProduct
44  {
        3 references
45      string GetAllProducts();
46  }
47
    0 references
48  public class ProductAdapter: ProductManager, IProduct
49  {
        3 references
50      public override string GetAllProducts()
51      {
52          string returnXml =  base.GetAllProducts();
53          XmlDocument document = new XmlDocument();
54          document.LoadXml(returnXml);
55
56          return JsonConvert.SerializeObject(document,
57              Newtonsoft.Json.Formatting.Indented);
58      }
59  }
60
```

Figure 6.7: *Product Adapter*

In this example, an adapter class called "**`ProductAdapter`**" was created to manage the conversion from XML to JSON. That means the client classes will use and call this adapter instead of the original class; considering the objective is to use only the return in JSON format in that context.

This pattern is used mainly for database operations once the source has stored data in a specific format. The target projects need to consume and receive this data in a format understandable by a programming language such as C#, which expects to handle the return in JSON, Datatable, string, and other types known by the language.

Observer pattern

This pattern is extensively used in C# language, and it can be easily found in the .NET platform in cases where user interfaces need to be notified about changes in their state based on backend changes, such as desktop applications and Blazor components. The main objective of this pattern is to allow objects to subscribe and receive notifications from another class or provider. This provider must implement the interface **IObserver** to achieve this goal. Additionally, this pattern is helpful in all cases where the system needs to use push notifications, which are common in mobile applications.

For example, in an Observer pattern implementation, imagine a scenario where an online store exists and every user trying to buy a particular product needs to have updated information about the product on their screens, such as price, product description, and availability. Each operation in the management system that changes the store's product state must notify all the customers of a transaction in progress. That validation is essential for this kind of business and aims to avoid unexpected behavior of the system and legal problems. The Observer pattern can be used to achieve its objective, and it is mandatory to have at least two classes in this process: an observer class and another to be observable.

Considering the context of this example is related to Product, we must create a notifier class using the **IObserver** interface, passing the underlying Product class to the interface, as shown in *Figure 6.8*:

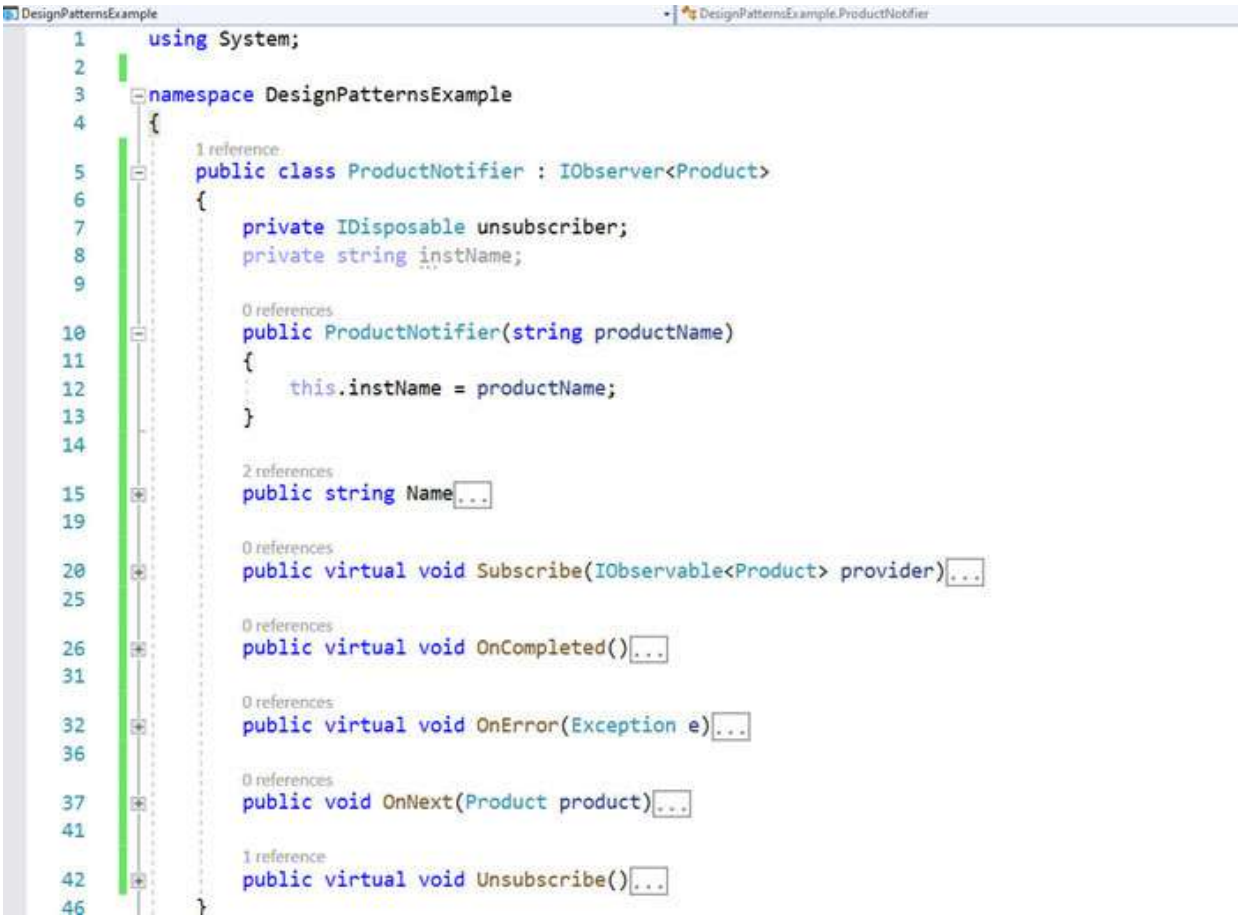


Figure 6.8: *Product notifier class*

Using the interface, **IObserver** requires implementing the Subscribe and Unsubscribe, **`OnNext`**, and **`OnCompleted`** methods, which allows us to specify custom implementation regarding these operations. The **`Subscribe`** method must receive a provider object as a parameter considering the provider represents the class responsible for implementing the notification operations, as shown in *Figure 6.9*:

```
0 references
public virtual void Subscribe(IObservable<Product> provider)
{
    if (provider != null)
        unsubscriber = provider.Subscribe(this);
}

0 references
public virtual void OnCompleted()
{
    Console.WriteLine("The price transmission was completed to {0}.", this.Name);
    this.Unsubscribe();
}

0 references
public virtual void OnError(Exception e)
{
    Console.WriteLine("{0}: The price cannot be determined.", this.Name);
}

0 references
public void OnNext(Product product)
{
    Console.WriteLine("{2}: The current price is {0}, {1}", product.Price);
}

1 reference
public virtual void Unsubscribe()
{
    unsubscriber.Dispose();
}
```

Figure 6.9: *Methods of the observer class*

Usually, these operations are necessary to change the state of the components in a screen regarding the class properties or could even make log operations and notify external systems. On the other hand, the class responsible for receiving the notification would have a similar implementation as shown in *Figure 6.10*:

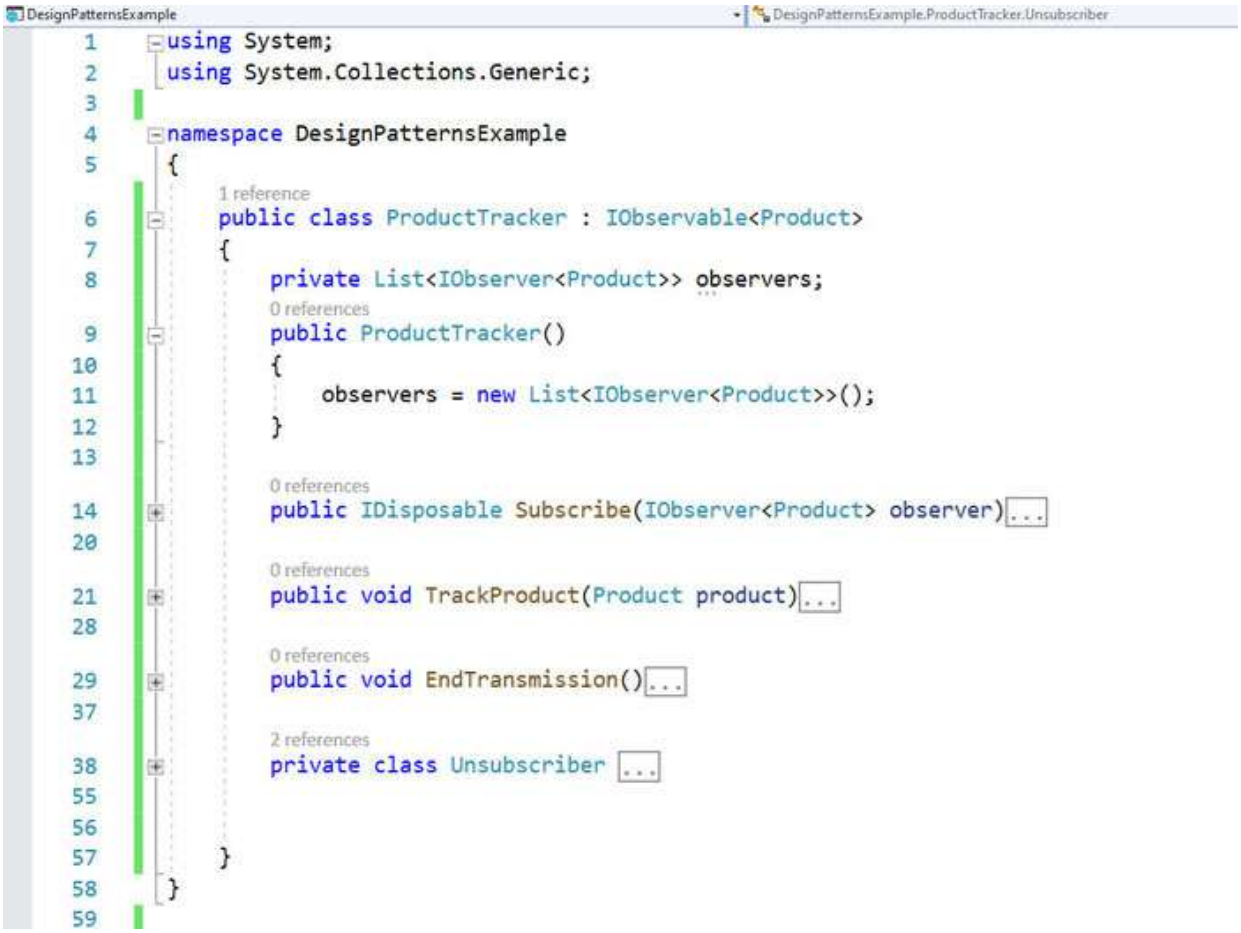


Figure 6.10: *Observable class*

The class must implement the **IObservable** interface, and by requirement, the method Subscribe must receive the related observer class as a parameter and implement the routine to add the observers the new observer, as shown in *Figure 6.11*:

Figure 6.11: *Subscribe method*

As seen during the section, the observer pattern is recognizable in legacy project, and many libraries used with the.NET platform must adhere to the same pattern and share similar interfaces once it has been implemented. The same design pattern can be found in other languages and platforms, and you will have the opportunity to be in touch with this pattern using JavaScript and SignalR later in this book.

Builder pattern

The Builder pattern has the purpose of splitting and simplifying the creation of complex objects by following the good practices of separation of concerns between the representation of the class and the logic behind the creation of the object. This pattern helps provide a better understanding of the implementation for developers. It makes it possible to easily create a suite of

tests to validate the implementation; considering the fact, all the complex operations are separated into clear and small steps.

Following the similar example that was already given in this chapter related to products and related operations in an online store, imagine the scenario where a product in the system should be a result of information provided for different systems for a single product:

- The internal store system should provide the name.
- An external system maintained by partners should provide the price.
- The photos of the product should be provided by another external database bought from another company.

Creating an object in this scenario is complex once the necessary information comes from three different sources and evolving infrastructure operations such as different API calls, database connections, and other responsibilities. To simplify and separate the implementation of all these distinct parts, the build pattern can be used, as shown in *Figure 6.12*:

```
using System.Net.Http;

namespace DesignPatternsExample
{
    public class OnlineProductBuilder : ProductBuilder
    {
        private Product _product = new Product();
        public override void BuildName()
        {
            HttpClient client = new HttpClient();
            this._product.Name = client.GetAsync("/getProductName").Result.Content.ReadAsStringAsync().Result;
        }

        public override void BuildPhotos()
        {
            HttpClient client = new HttpClient();
            this._product.Photos = client.GetAsync("/getProductPhotos").Result.Content.ReadAsStringAsync().Result;
        }

        public override void BuildPrice()
        {
            HttpClient client = new HttpClient();
            this._product.Photos = client.GetAsync("/getProductPrice").Result.Content.ReadAsStringAsync().Result;
        }
    }
}
```

Figure 6.12: *ProductBuilder class*

In this example, three distinct methods were created with a unique and clear responsibility. In the creation of the product object, each method will be called individually in a specific order if there is some requirement associated with the order, as shown in *Figure 6.13*:

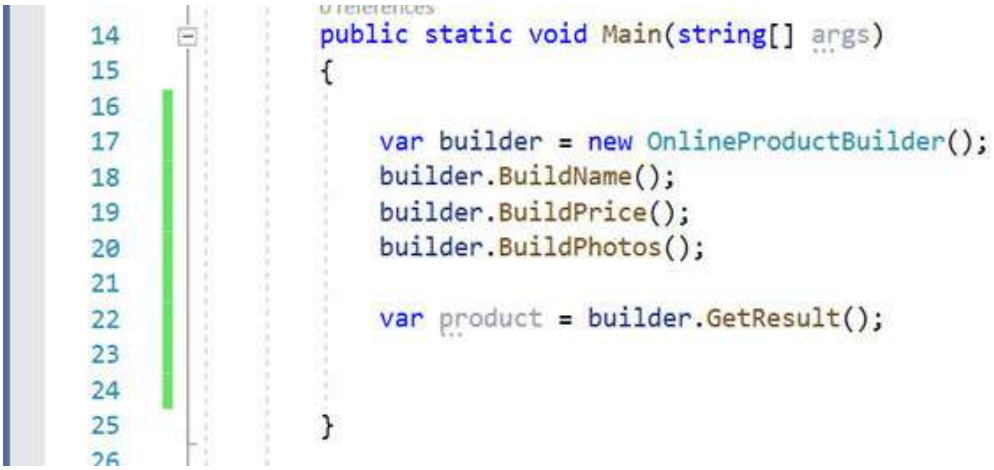


Figure 6.13: *Creation of objects using Builder*

In this case, each method responsible for creating a specific part of the object is called in sequence. In that specific example, an object of the type of Online Product was created, which has its way of populating the product's properties. Each type of product can have its own builder class where it can customize all the individual operations.

Factory pattern

The Factory pattern reduces the dependency between components in the software, encapsulating the complexity of classes, operations, and distinct types to other classes that consume specific resources. This concept is hugely used in the .NET Core platform, and the implementation is intrinsically related to the concept of dependency injection. One of the most significant improvements in projects based on .NET Core comparison to .NET Framework is the possibility to register classes and services to be used as Factory in the startup class and inject these objects to other classes across the software without exposing it.

For instance, imagine a scenario where there is an external service that would be used in many parts of the project. There are two main objectives and requirements, such as having the possibility to use mockups to test the components associated with the external service and to avoid the creation of the same object several multiple times in other classes. To achieve this objective, in Asp.Net Core applications, it is possible to specify services in the file Startup.cs class and use single or multiple instances of the object in Controllers, as shown in *Figure 6.14*:

```
2 references
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    1 reference
    public IConfiguration Configuration { get; }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();

        services.AddSingleton<ExternalService>();
    }

    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)...
}
```

Figure 6.14: *Factory Pattern*

In this example, every class in the software that contains in the constructor a parameter with the same interface as the class **`ExternalService`** will receive by dependency injection the single instance of the external service object, as shown in *Figure 6.15*:

```
namespace DesignPatternsExample.Controllers
{
    1 reference
    public class HomeController : Controller
    {
        private readonly IExternalService _externalService;

        0 references
        public HomeController(IExternalService externalService)
        {
            externalService = _externalService;
        }

        0 references
        public IActionResult Index()
        {
            return View();
        }

        0 references
        public IActionResult Privacy()
        {
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        0 references
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Figure 6.15: *Dependency Injection*

Considering an interface is being used for injecting the external service in the Controller, it is possible to create a mockup class in case of an integration test is required. This feature is used mainly for database connections, HTTP requisitions, stream services, and other operations that depend on external resources.

Conclusion

As seen in this chapter, the design patterns are broadly used in .NET projects using C# language and object-oriented paradigm and provide exciting and useful features and benefits such as solutions to build robust enterprise applications with the applicability of the best practices of software development used in the market by big companies in terms of software architecture and testability that allow us to reduce the dependency between components in the projects and to make the maintenance easier.

Considering that design patterns are used in many libraries and projects on a global scale, it is mandatory knowledge to understand legacy projects and simplify the development of new projects that demand high quality in terms of practices and reusability, reducing the costs of projects, and minimizing the risks regarding software maintenance.

In this chapter, you learned how to use the most common design patterns in .NET projects such as Singleton, Builder, Factory, Observer, Façade, and Adapter by applying their concepts to solve different problems in software development. Additionally, you had the opportunity to understand how to take the benefits of building testable, reusable, and extensible applications using advanced software architecture concepts using design patterns in the C# language.

In the next chapter, you will have the chance to learn the basic concepts of C# language, such as operators, loops, and iterations, to get familiar with the fundamental logical commands in C# language and prepare for the future content in this book.

Points to remember

- The design patterns are primarily used in the market to solve common problems in software development, representing the knowledge shared.

- The Factory and Single patterns can be used in .NET Core projects combined with dependency injection.
- Developers must easily recognize the use of any design pattern in legacy projects.

Multiple Choice Questions

1. **Which design pattern can be used to use the same instance of an object across the system?**

    a. Façade

    b. Singleton

    c. Adapter

    d. None of them

2. **Which pattern is used in .NET Core applications combined with the concept of dependency injection?**

    a. Factory

    b. Object-Oriented Programming

    c. Singleton

    d. Builder

3. **Which pattern is helpful to create complex objects in small steps?**

    a. Polymorphism

    b. Adapter

    c. Builder

    d. Façade

Answers

1. **b**
2. **a**
3. **c**

Questions

1. Explain the main advantages of the Façade pattern.
2. Why should we use design patterns?
3. What is the correlation between Factory pattern and Dependency Injection?
4. Explain the purpose of the Façade pattern.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7

Operators, Loops, and Iterations in C#

Introduction

In this chapter, we will walk you through the fundamental concepts of programming in C#, giving you more familiarity with the language, its primary operations, and instructions that will help you understand what you need to make progress in the following sections of this book, such as the new features introduced to the C# language and .NET Core.

You will learn to create and work with variables, operators, and logical and conditional statements. Additionally, you will have the opportunity to have practical experience in implementing basic programs in C#.

Getting familiar with the basic concepts of C# is essential to understand how to apply the necessary structures in real scenarios in enterprise projects.

Structure

In this chapter, we will discuss the following topics:

- Object Types in C#
- Basic operations in C#
- Loops and iterations in C#

Objectives

After studying this chapter, you should be able to apply the most common operator using C# language in basic programs, know the object types in C# language, and understand the difference between the loop instructions available in C# language.

Object types in C#

The best way to learn any programming language is to create real applications for practising and apply the knowledge you have in real scenarios. First of all, fundamental concepts apply to any programming language, including C#

languages, such as variables, statements, loops, blocks, and iterations. In the following sections of this book, you will have the opportunity to create multiple C# applications, and in the meantime, all the basic concepts will be explained.

As a strongly-typed language, in C# language, you must specify the type of any variable created in the program, indicating the type of values that will be stored on it.

Tip: Strongly-typed languages are the languages that require all the variables with a specific type when it is created. That means if you create a variable to store only numbers and try to store text content in it, the compile will show an error indicating that the variable's value does not correspond to the variable type. In computer programming, there are weakly typed languages, which are not pre-compiled, but the type is checked dynamically once the application runs.

A variable in a programming language can store simple data structures such as numbers and strings but can also contain complex data such as files, lists of custom objects, and much more. Their correct use represents an important point in software development, as they are closely related to how we interact with the memory in any application. They can represent a good measure of the software's readability in terms of coding. It is essential to follow good practices and conventions related to variable names, not only for this specific small part of the software but for other structures used in C#. Always use meaningful names across the system, applying the Camel Case pattern for classes (MyBpbBook) and Upper Case for constants (for example., `MAX_ITEMS_ORDER = 3`). Make your code easy to read by others and even by yourself.

Figure 7.1 represents a simple creation and value assignment for a C# variable:

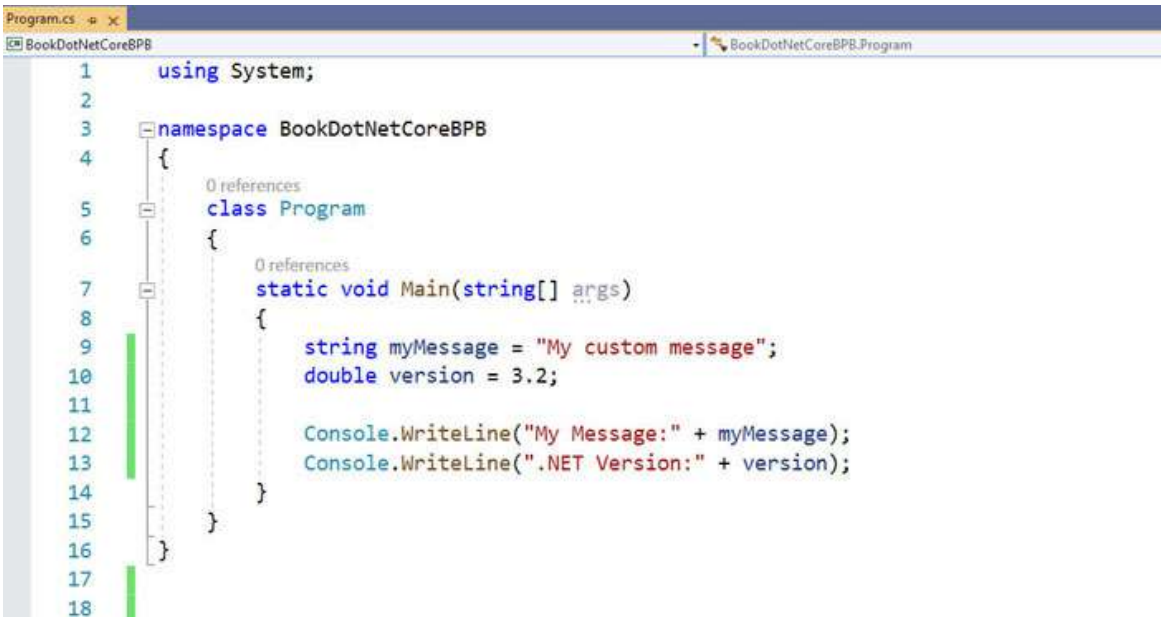


Figure 7.1: *C# variables*

Lines 9 and 10 represent two different variables in C# from distinct types. The first is a string variable, storing the content “My custom message.” The value must be included in quotes because that is a pattern for any string in the C# language. Referencing the second variable (line 11); it is storing a decimal value (3.2), and because it is not a string, the value does not need to use for quotes. The variable should be named according to its purpose and can be used in other parts of the program within the same context, such as the same method scope or other methods, in case this variable is created as a global one.

There are many types of variables in C# language, and each one has a specific reason to be used. The following topics contain the most popular types and their detailed explanation with examples:

- **Int:** This type was created to store integer numbers, positives, or negatives. As happens with any variable type, there are limits (maximum and minimum values) that can be stored on it. In that case, the minimum value is -2,147,483,648, and the maximum value is 2,147,483,647. This limitation helps to prevent security issues and to optimize memory usage. Each time a variable is created in the program, space is reserved in the memory to store the value. Therefore, choosing the suitable variable type for any variable used in the system is highly recommended.
- **Bool:** It has the purpose of storing Boolean values (true or false). Additionally, it is possible to assign null values to variables of this type using the keyword “?” after the type, as shown in *Figure 7.2*:

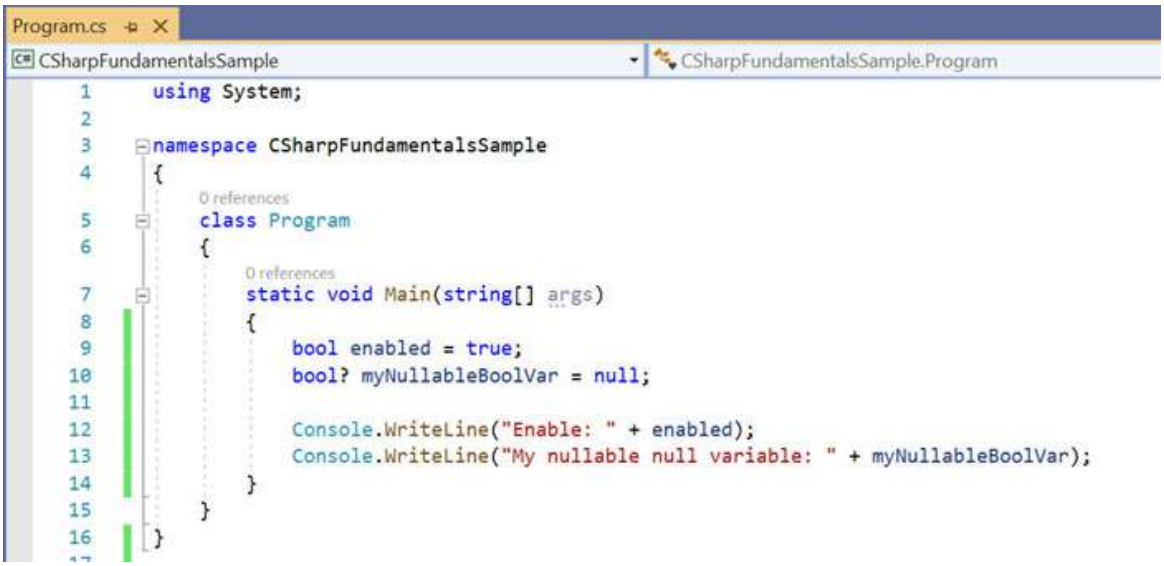


Figure 7.2: *C# bool variables*

Line 9 represents a bool variable storing a "*true*" value, and line 10 consists of a nullable bool variable.

- **String:** this type was made to store a sequence of Unicode characters and requires quotes around the value. It will indicate to the compiler that it is effectively string content.
- **Byte:** This is a type used for storing unsigned integers values that start with "0" (minimum value) and "255" (maximum value). It is commonly used to manipulate file bytes in arrays of that type (byte) and low-level programming.
- **Char:** This was created to store a single Unicode character, different from the string type that stores a sequence of those. The utilization of this type helps allocate less space in the memory if the content is just a simple Unicode character. If the quantity of characters in a variable is pre-defined and it would have just one Unicode character, choose this type of variable instead of string type.
- **Object:** This is the most basic type in C# language since all the other types inherit from it. It is possible to assign values of any type to it. It was made to be completely generic and is primarily used in dynamic types, mainly for assignments when the content has an unknown type.
- **Float:** This is used to store numbers with precision between 6 and 9 digits, and the maximum size is just 4 bytes.
- **Double:** This stores numbers with precision between 15 and 17 digits, and the maximum size is 8 bytes.

- **Decimal:** This stores numbers with precision between 28 and 29 digits, and the maximum size is 16 bytes.
- All these variable types in C# are considered **primitive** types, representing the most straightforward structure in this programming language to store data. Each type uses a specific size in memory after its creation, and the .NET Core platform handles the clean-up of them in a high-performance way once they are considered reference types.

Tip: In C# language, there are differences between variables of Value Type and Reference Type. The primitive types (int, string, bool, and so on.) are considered value types once stored in their memory space. On the other hand, the Reference Type has a reference placed in the memory, and its underlying value is placed in a different memory space. Being familiar with these differences is important when the scenario requires high demand and better performance.

If the incorrect value type is assigned to a variable of a particular type, the compile will show an error, and the program will not run. Using a strongly-typed language such as C# is a considerable advantage because it is possible to figure out programming issues earlier compared to the languages that accept only dynamic types whose values are verified in runtime. The following image contains samples of the creation and assignment of each variable shown and explained so far, focused only on primitive types and the most frequently used types in C# programming language, as shown in *Figure 7.3*:

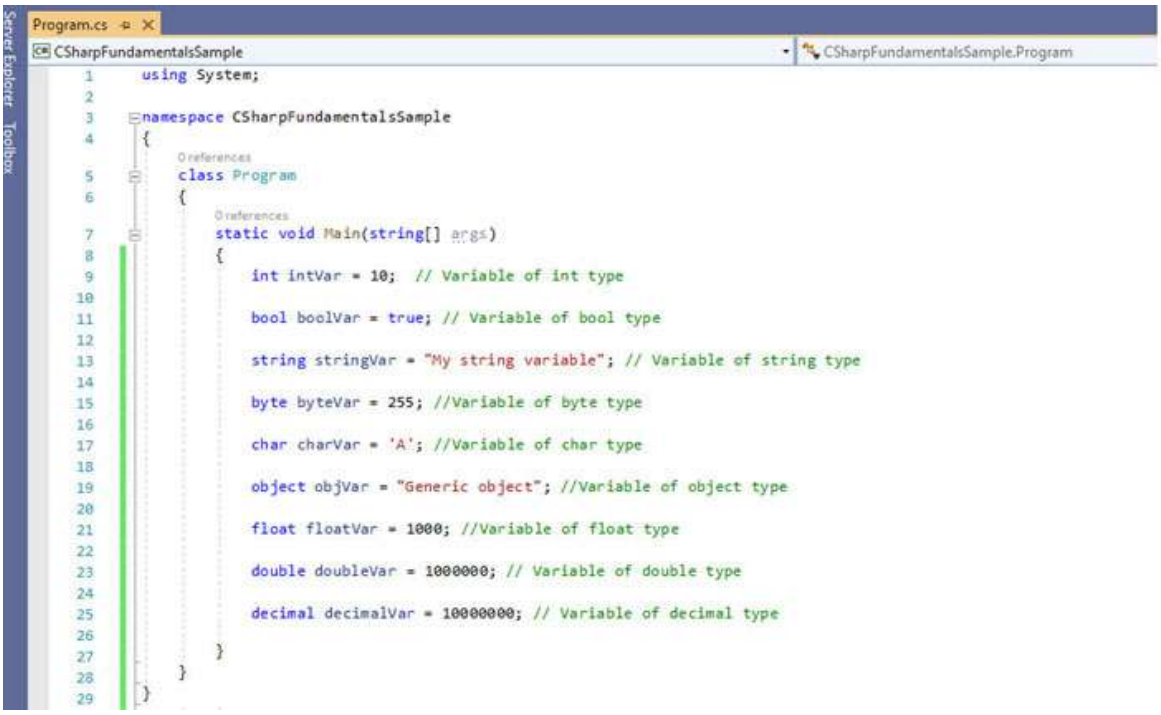


Figure 7.3: *Primitive types in C#*

Meanwhile, the Reference Type variables are used to store more complex data and are stored differently in the memory from Value Type variables. Apart from primitive types, it is possible to create variables of your own custom types or even of other available native complex types in C# language such as List, Array, and others. Essentially, the syntax to create reference types is the same, declaring the type before the variable's name.

Note: Despite string and object being primitive types, they are considered reference types in C# language simultaneously as class, interface, and delegate types. Therefore, there is no mandatory relation between value types and primitive types.

The following topics contain the most popular native reference types and their detailed explanation with examples:

- **Array:** This type has the purpose of storing a collection of objects, having multiple values in a single variable, and optimizing the time in manipulating a considerable amount of values using multiple distinct variables. To create an array, you must declare the variable type followed by the square brackets, and the values can be assigned separately from each other by a comma. As shown in *Figure 7.4*, to manipulate the values in the array, you must use any

available iteration statement in C# language (loops, foreach, and for), which will be explained in detail in the upcoming chapters of this book:

Figure 7.4: *Arrays in C#*

- **List:** This is a complex object that allows us to manipulate elements as a list and access those values by index. This type contains many extension features to search list elements, sort, and make various operations. It is one of the most used types in C# because many programs have in common the need to manipulate a list of elements coming from databases and other sources. When the List has been created, the type of the elements should be defined, and the values can be assigned using the method “Add,” which belongs to the List class. This type supports a wide variety of operations, and those operations are the main difference between that type and array variables. Both of them can get similar results in manipulating a list of elements. Still, the complex type List contains more advanced options to insert, add, remove, and search items in the list without explicitly referring

to their indexes. It prevents errors in dynamic operations once arrays always should have a specific range of values. *Figure 7.5* has an example of the List variable type and the way to assign elements to it:

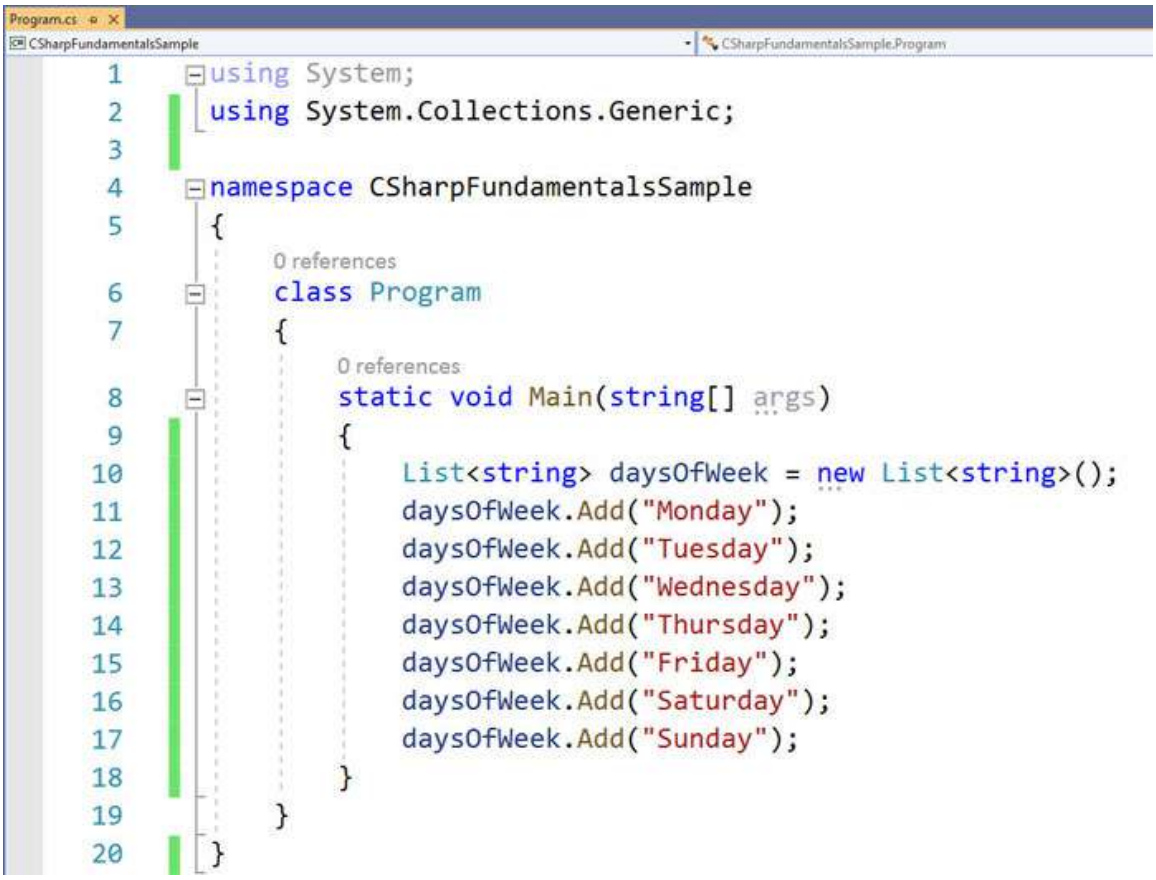


Figure 7.5: *Lists in C#*

- **Class:** In the C# language, it is possible to create your own types, as it is object-oriented. There are many particular concepts related to class, but to get started with classes, you must know that they are one of the reference types in C# language; their correct usage and construction are the most important part of any software using the C# language. It is possible to define properties, visibility, constructors, and other inherent concepts in object-oriented programming through classes. Understanding it as one of the available types in C# is essential. The following image contains an example of a class called “Book,” which has three properties: ID, Title, Creation, Author, Summary, and ISBN. You must realize that the types of properties correspond to primitive or complex types, depending on the kind of information that should be stored on them. The “Author” property represents a complex type of “Author” class; therefore, it refers to another entity, as shown in *Figure 7.6*:

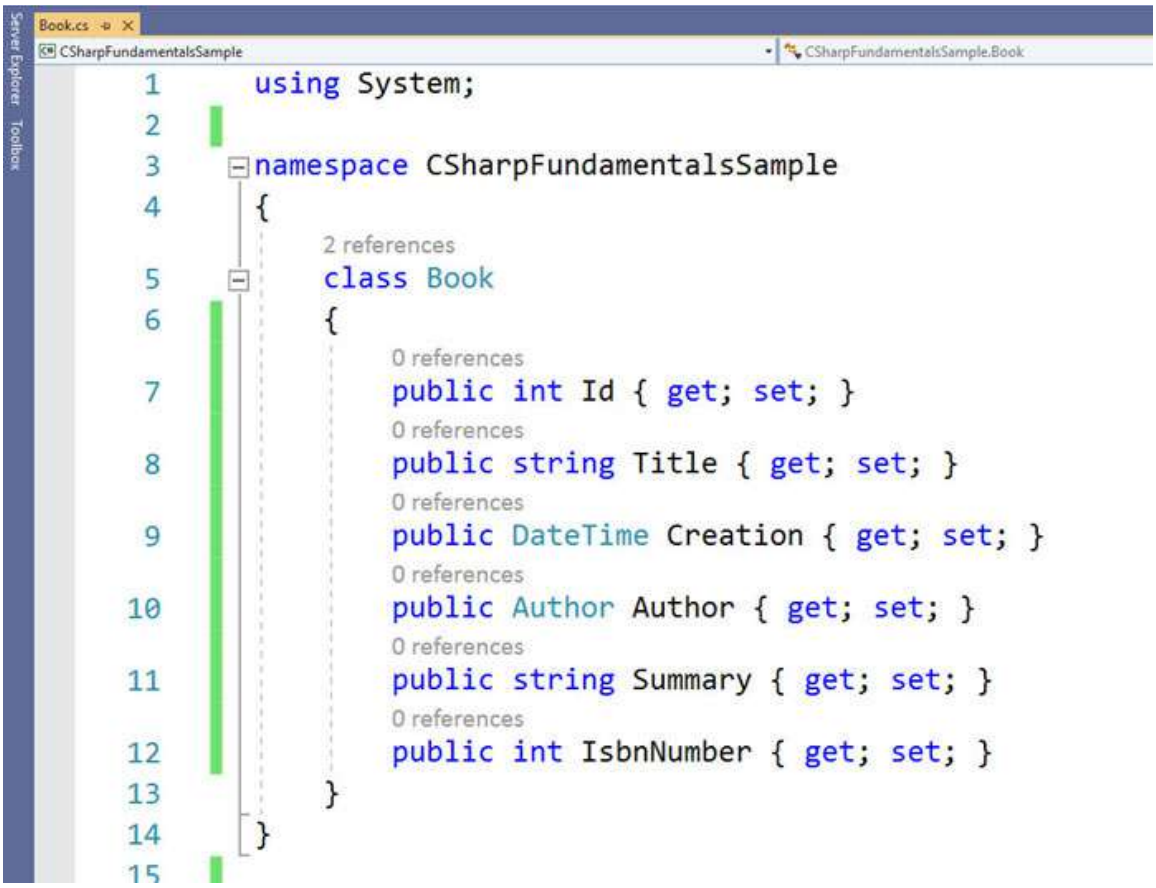


Figure 7.6: *Class in C#*

- **Other types:** There are other reference types in C#, such as delegates and interfaces, which will be explained in detail in the upcoming chapters. They involve more advanced concepts, and many examples of them can be found in this book.

Understanding the most basic types will allow you to get started with performance enhancements in C# language and help you follow the best coding practices.

Loops, operation, and iterations

There are many ways in C# language to manipulate a list of objects and execute recursive logical programming, to meet specific requirements and significantly reduce the number of necessary code lines for certain implementations that need to make operations many times. In the following topics, you will have the opportunity to walk through the most common statements to execute repetitive operations, and you will also understand the difference between them.

While statement

The "**`while`**" statement can execute operations until the pre-defined condition is not satisfied (false). The program will test the condition at the beginning of the loop, and if the result is true, the loop will still be executing. Its use is pretty similar to other statements in C# language for loop operations, but the syntax used for the condition is clear enough for the developer who is trying to interpret and debug the code. In the following image, there is an example of a while statement, which has a condition to keep running till the condition is not satisfied (number greater than 25), as shown in *Figure 7.7*:

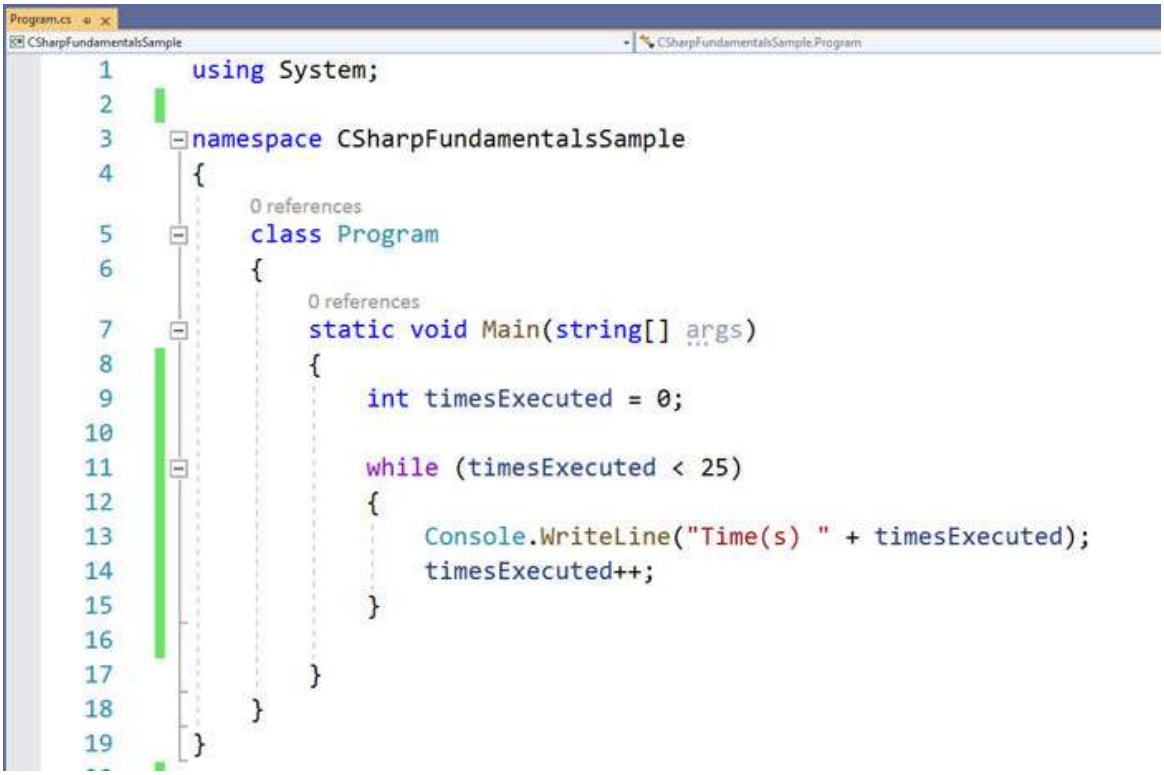


Figure 7.7: *While in C#*

The preceding code has the purpose of printing the message "Time (s)" followed by the time of loop execution, and it should run until the number of times. Therefore, the operation will happen 25 times, considering the initial value of the variable "**`timesExecuted`**" is "0," and the variable is increasing by one number on each iteration, as is seen in line 14.

Do-while statement

The do-while instruction is very similar to the regular "while" statement, except that the logical code within the block will be run before the code that tests if the

pre-defined condition is satisfied. The following image contains the exact same purpose and logic programming of the previous example but using do-while instead, as shown in *Figure 7.8*:

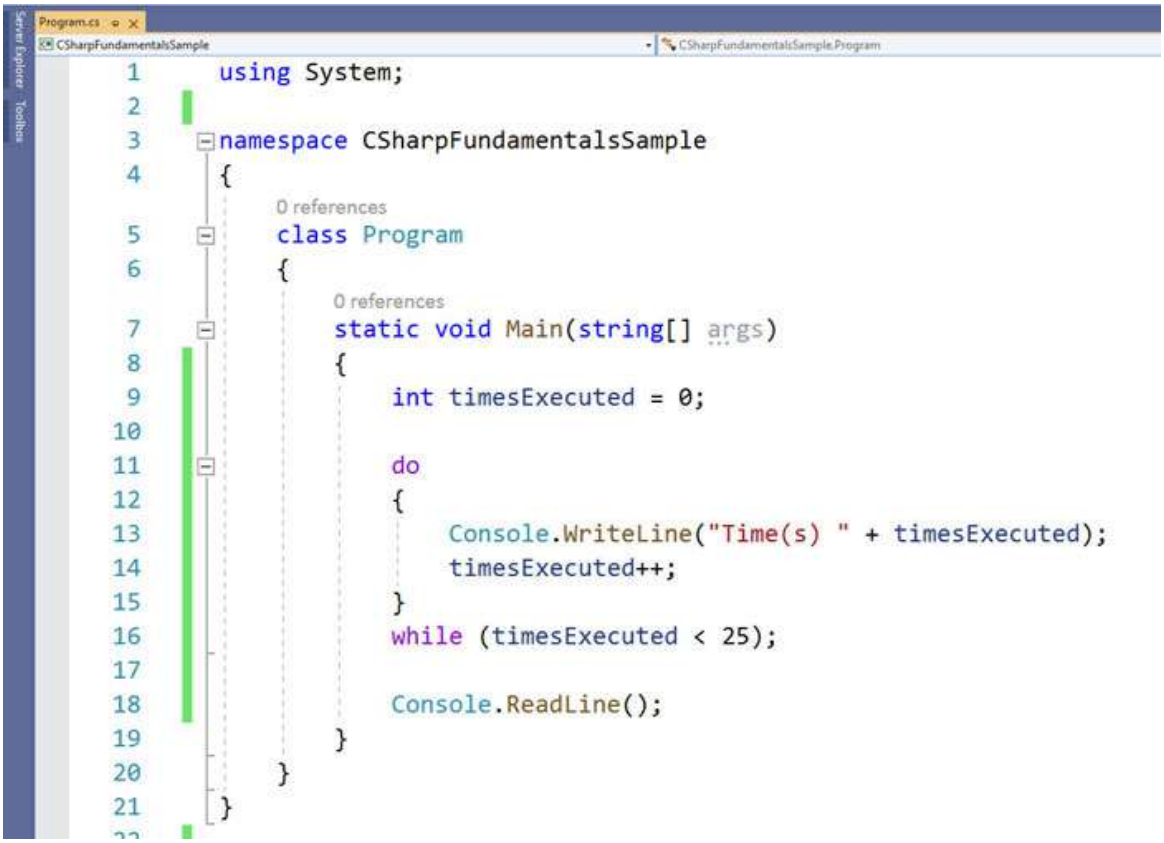


Figure 7.8: *Do-while in C#*

This type of loop is proper when a particular operation must be done before the routine tests the condition, representing the main difference from the while statement.

In this simple routine, the initial value of “timesExecuted” is “0,” and the body is executed without testing the condition for the first time. Within the body, the value of “timesExecuted” is one number, which means that the routine will be executed 25 times. In specific scenarios, the condition tested by the “while” could never be satisfied, and it will cause an infinite loop.

For loop

There are many ways to get similar results in a programming language, and this premise applies to loop operation in C#. Apart from “while” and “do-while” statements, it is possible to make similar operations using a “for” loop, which allows us to execute repetitive operations when the number of times for the loop

is known before it starts to run. To use this type of loop, you must create a “for” statement following these requirements:

- A variable that contains the initial value for the iterations should be created inside the “for” statement.
- The condition to be tested and satisfied should use the variable created for the initial iteration.
- The times of the executions should be incremental and pre-defined.

To clearly understand this type of loop operation, you must see *Figure 7.9*, which contains similar purposes to the previous examples:

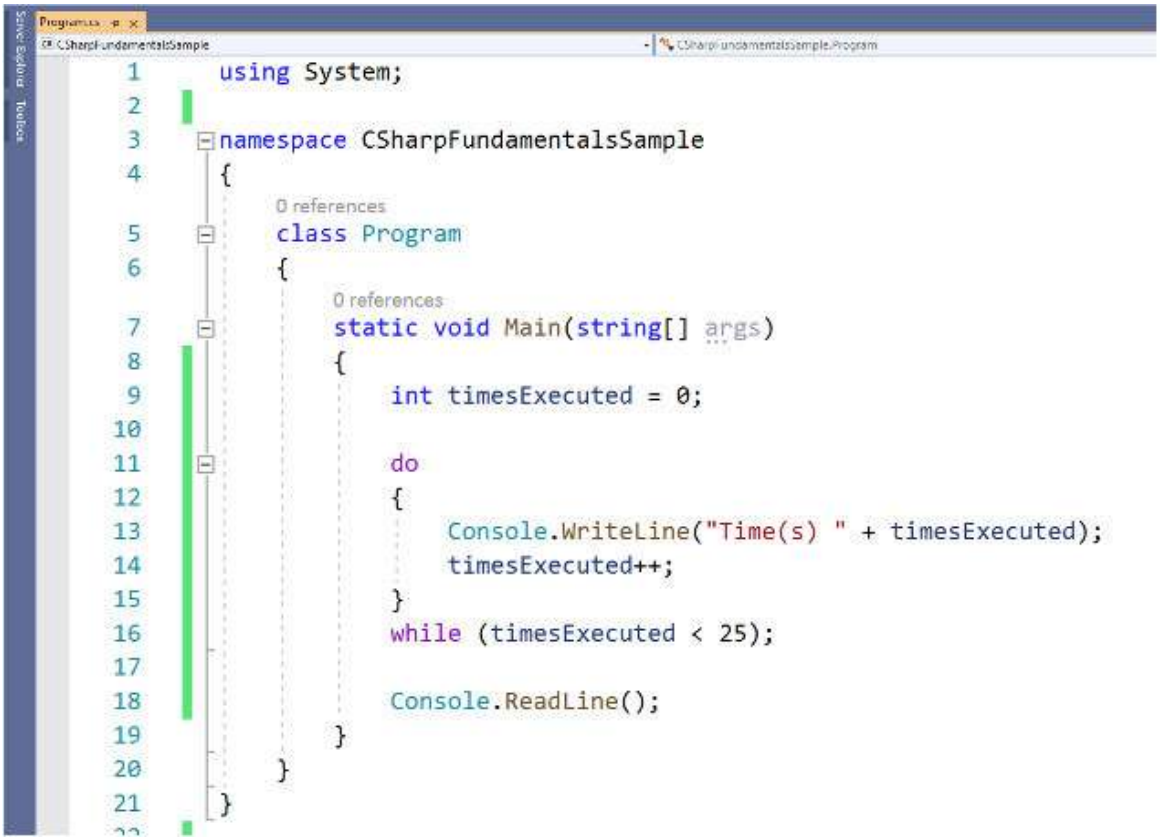


Figure 7.9: *For loop in C#*

As shown in *Figure 1.16*, the variable “timesExecuted” is declared inside the instruction, and the incrementation process occurs after the condition that needs to be satisfied. If you compare it to the previously explained loops in C#, it seems to be simpler and easier to read than the other ones, but it has the same purpose: printing content 25 times on the screen. Therefore, there are many ways to reach the same target in logical operations, and you must choose the suitable one for your own scenario in enterprise solutions.

Foreach statement

The "**foreach**" loop can be used to execute operations in a list of elements and in any type which implements the **IEnumerable<T>** interface in C#. *Figure 7.10* represents how it works to execute a block of code for each individual item in a list:

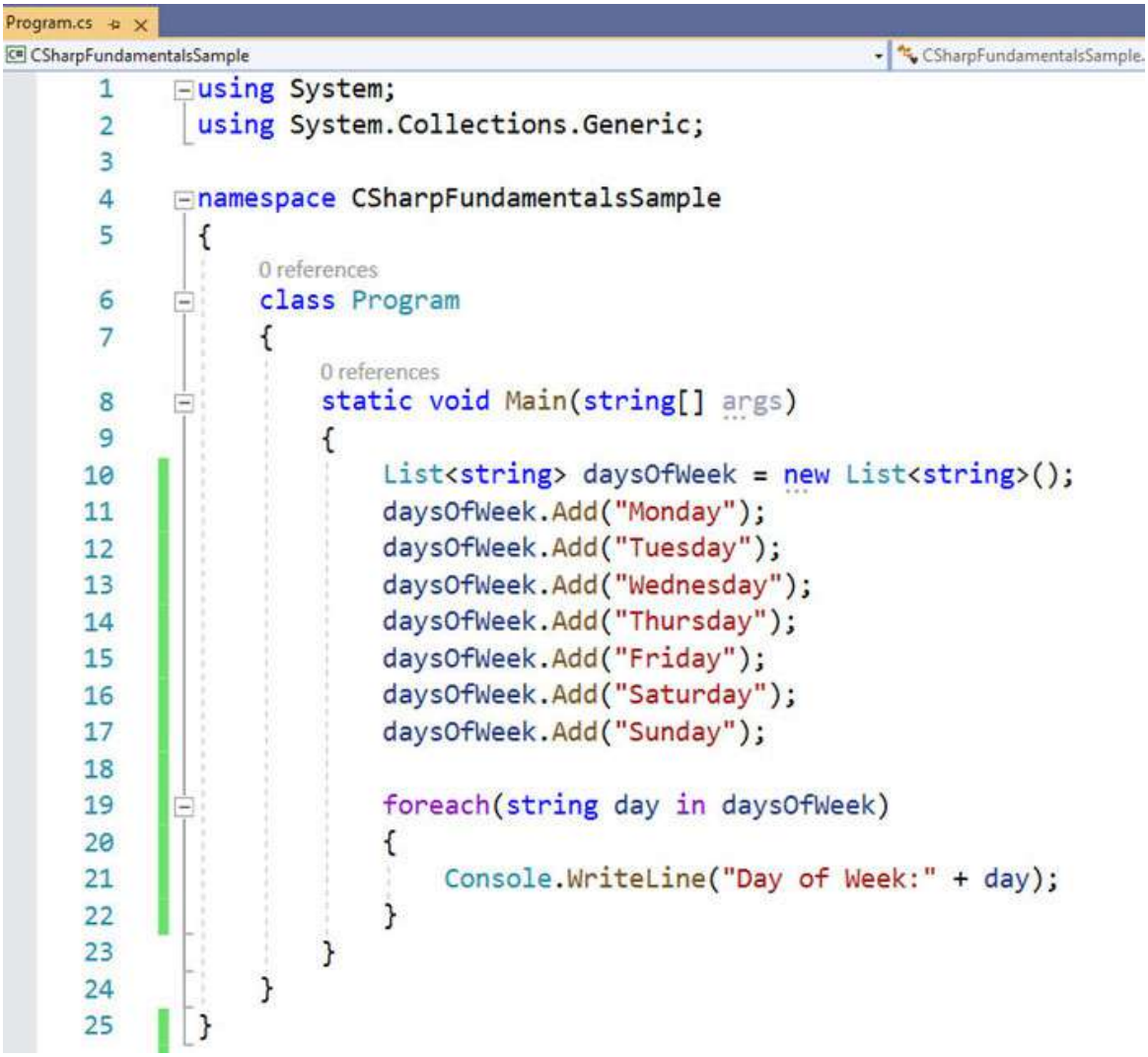


Figure 7.10: *Foreach in C#*

As seen in the foreach statement sample, in line 10 is created a List of string items, and between lines 11 and 19, it is populated with the days of the week, from Monday to Sunday. Finally, a "foreach" instruction is executed once for each item on the list. That means the block of the code which prints the "Day of Week" runs seven times. In general, loop implementations have the advantage of writing less code in case it contains repetitive logical operations. In this example, instead

of specifying the **`Console.WriteLine`** instruction seven times (one for each day); it was specified just once.

Operators

Operators are used mainly in mathematical operations and programming languages to make arithmetic operations, compare values, and specify conditions for boolean values. The C# language contains a vast number of operators, and in the following topics, you will have the opportunity to walk through the main types, their usage, and helpful examples.

Arithmetic operators

This type is used to make addition, subtraction, division, multiplication, and other popular and conventional arithmetic operations. It is possible to specify these operations using variables or oven using values directly, as shown in *Figure 7.11*:

```
Program.cs
CSharpFundamentalsSample                    CSharpFundamentalsSample.Program
 1   using System;
 2
 3   namespace CSharpFundamentalsSample
 4   {
         0 references
 5       class Program
 6       {
             0 references
 7           static void Main(string[] args)
 8           {
 9               int number1 = 50;
10               int number2 = 25;
11
12               int totalSubtraction = number1 - number2; // 25
13               int totalAddition = number1 + number2; // 75
14               int totalMultiplication = number1 * number2; //1.250
15               int totalDivision = number1 / number2; // 2
16
17               Console.WriteLine("Total subtraction: " + totalSubtraction);
18               Console.WriteLine("Total addition: " + totalAddition);
19               Console.WriteLine("Total multiplication: " + totalMultiplication);
20               Console.WriteLine("Total division: " + totalDivision);
21
22               Console.ReadLine();
23           }
24       }
25   }
```

Figure 7.11: *Arithmetic operator in C#*

As seen in the preceding image between lines 12 and 15, variables were created to store the total of the arithmetic operations of two numbers: variables number1 and

number2. It is important to note that the total variables should be created to support the underlying correct result operation. If at least one of the two numbers had a decimal value (for example, 25.4), the variable total should be from the decimal type or even a float variable, depending on the number of characters.

Assignment operator

The most common operator is the "**equal**" operator, which is used to assign value to a variable, as shown in *Figure 7.12*:

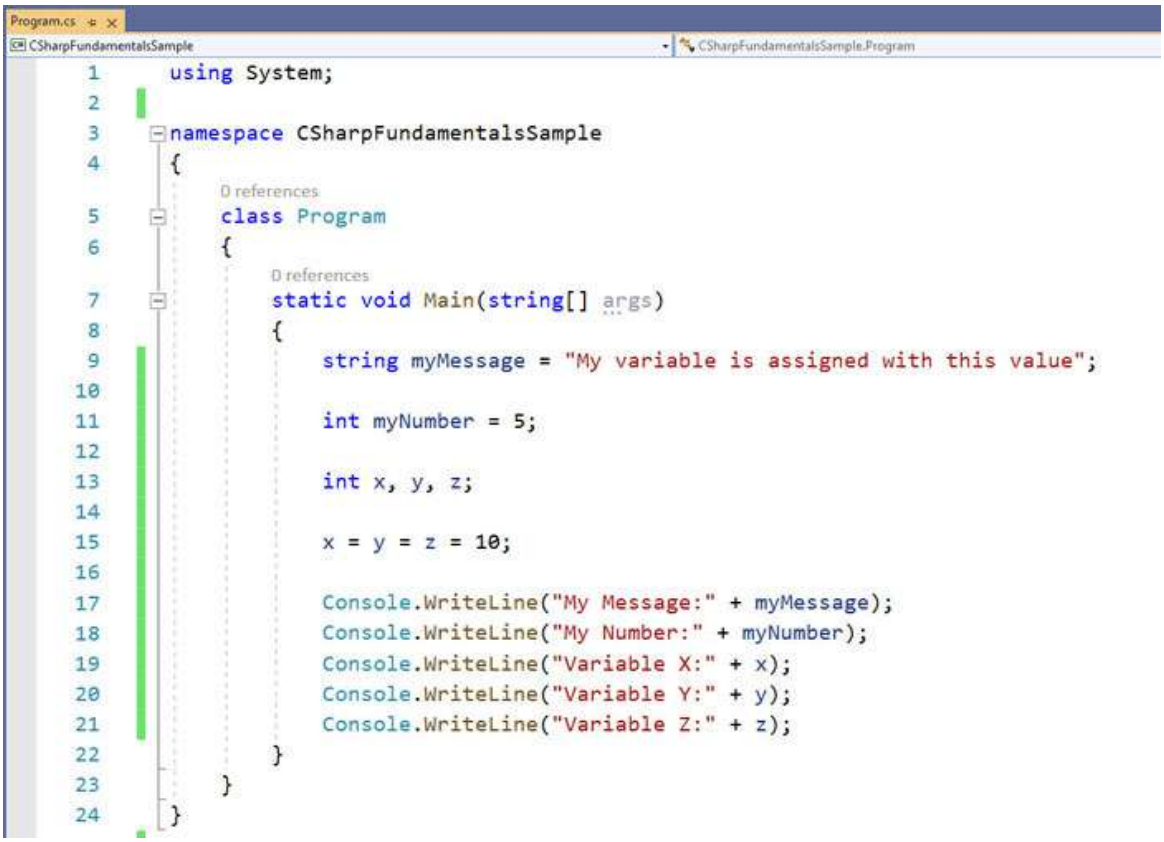


Figure 7.12: *Assignment operator in C#*

On line 9, a string variable called "**myMessage**" is created, and a value is assigned to it using the operator "=." The same process occurs in line 11, but with an int type. There is the possibility to assign a value to multiple variables in sequence using the equal operator, which means that all the variables will have the same value in the final. As in line 15, the int variables are assigned to the value 10. Therefore, the equal operator in C# language is not necessarily used only for arithmetic operations but also to set values. The variable always will be placed on the left side and the value on the right side.

Relation operators

One of the most common logical implementations in any programming language is boolean expressions, considering a system can have a routine that should follow specific logic based on certain conditions. To help with this target, the C# language contains many relational operators that allow us to compare values, from the most straightforward operation to the most complex ones, according to what is needed. The following table contains all the relational operators that can be used in C# language:

Name	Operator	Example	Expected result
Equal to	==	25 == 25	True
Not equal to	!=	25 != 10	True
Less than	<	25 < 15	False
Greater than	>	25 > 15	True
Less than or equal to	<=	25 <=25	True
Greater than or equal to	>=	25 >=25	True

Table 1.1: *Relational operators in C#*

Regarding the equals operator, in C# language, it is essential to know that to compare two values, two equal operators should be used. In that way, the compile will recognize it as a comparison and not as a value assignment. Also, the operator “!” can be used to reverse the boolean value for any statement, as shown in *Figure 7.13*:

```
int number1 = 25;
int number2 = 50;
bool resultOperation;

//EQUAL
resultOperation  = number1 == number2; // false

//NOT EQUAL
resultOperation = number1 != number2; // true

//LESS THAN
resultOperation = number1 < number2; // false

//GREATER THAN
resultOperation = number1 > number2; // true

//LESS THAN OR EQUAL TO
resultOperation = number1 <= number2; // false

//EQUAL
resultOperation = number1 >= number2; // true
```

Figure 7.13: *Relational operator in C#*

As seen in the preceding image, a variable called "**`resultOperation`**" is created to store the boolean result of each relational operation.

Logical operators

Logical operators are used to combine multiple relational operators in compound conditions. They are just boolean statements that return a true or false value. This kind of operator is largely used in loops and decision statements in logical implementations when one or more conditions in a sentence should be satisfied.

The "**`AND`**" and "**`OR`**" operators can be combined to make different comparisons depending on their order, and the statement can return different results.

For example, imagine a program with a specific operation that should be run only if the day of the week is Monday or Tuesday. The "**`OR`**" operator can be used in that case, as shown in *Figure 7.14*:

Figure 7.14: *"OR" operator in C#*

As seen in code line 12, there is an "if" statement containing two comparison expressions separated by the keyword "**`||`**," which is in C# equivalent to the "**`OR`**" operator. In that case, the code inside the "**`if`**" block will run only when at least one of the two conditions is satisfied: if the day of the week is "**`Monday`**" or "**`Tuesday`**."

Otherwise, if multiple conditions are tested, and all of them are true, the operator "AND" can be used. The keyword for that operator in C# is "**`&&`**" as shown in *Figure 7.15*:

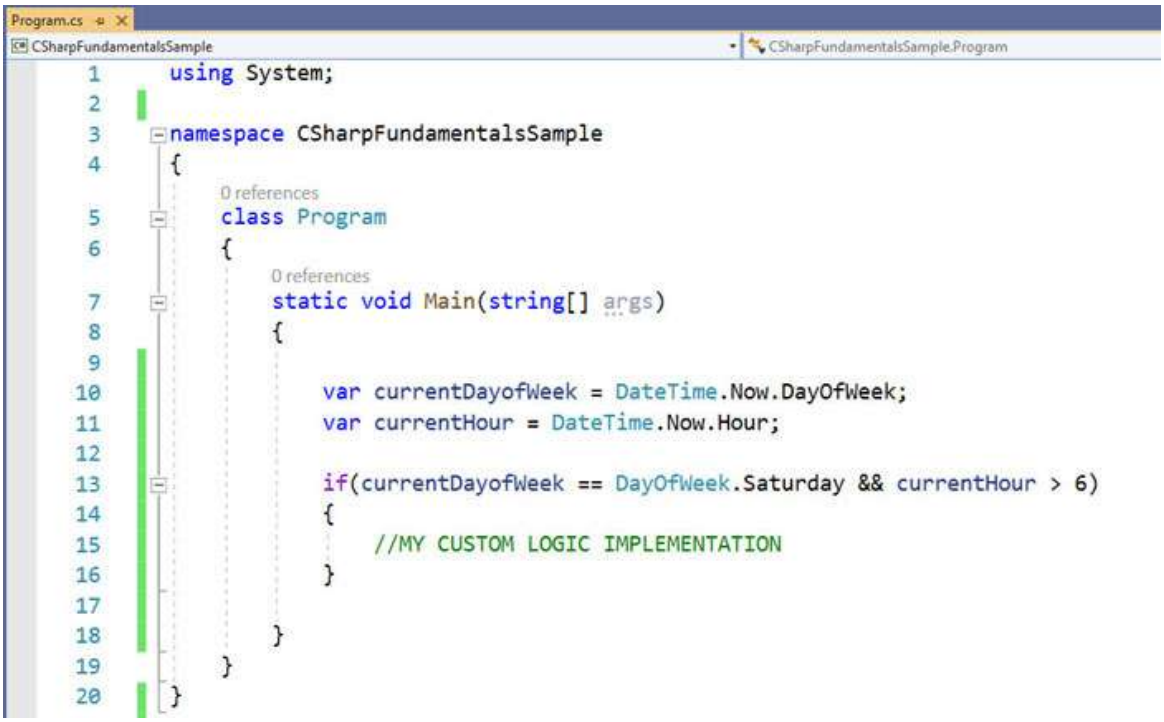


Figure 7.15: *"AND" operator in C#*

In the preceding example, the condition to execute the block of code inside the "**`if`**" statement will be satisfied only if the current day of the week is Saturday and the current hour is greater than 6.

Unary operators

The C# language has operators that have the purpose of making operations using a single operand. It saves time to write specific routines such as incrementing a number, inverting the value of a boolean value, decrementing a number, and others.

To add an extra number to an integer variable in C#, the operator "++" can be used, and it is possible to use the two minus operator to subtract in one a variable that contains a number value, as shown in *Figure 7.16*:

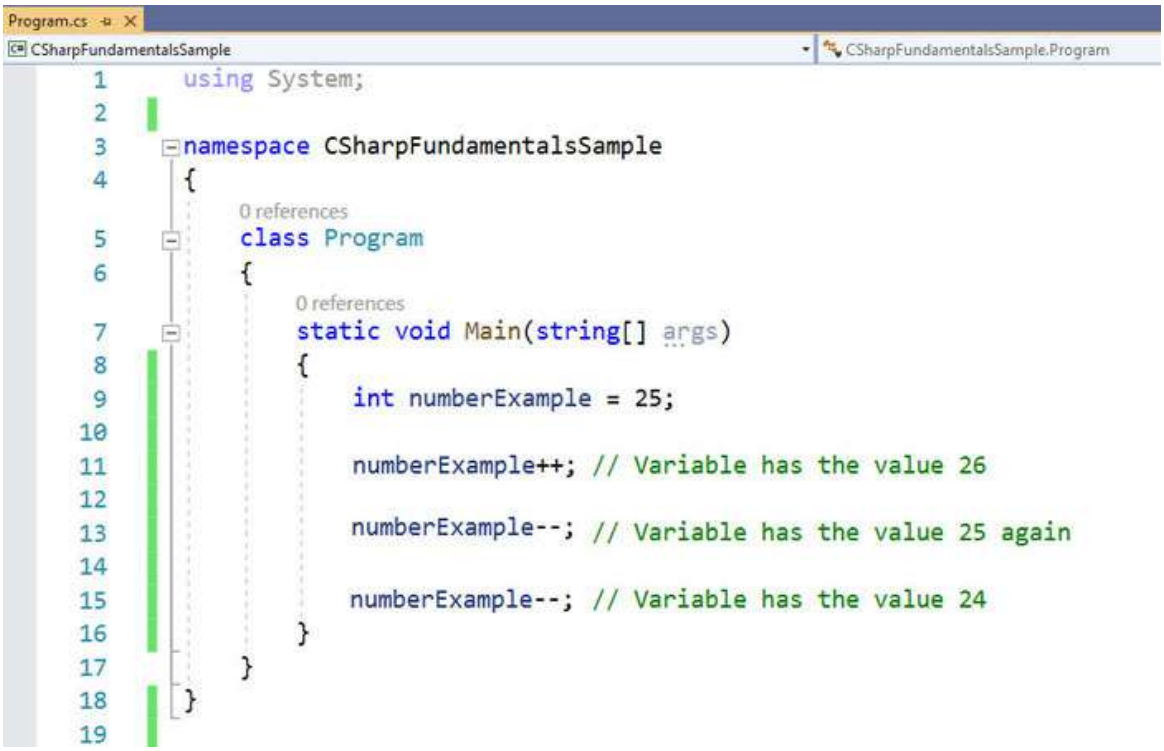


Figure 7.16: *Unary operators in C#*

The preceding implementation is the same as if it was adding one value to the variable using the arithmetic operator as **`numberExample = numberExample + 1`**. Undoubtedly, the unary operator syntax is much more optimized in terms of the amount of needed code to get a similar result.

Ternary operators

Another operator that helps to reduce the used line of codes in logical statements is the ternary operator that can replace "**`if`**" statements for comparison in case of the result of a specific condition is the assignment of a variable. As an example, in the following image, there is a variable called "**`message`**" from string type, and its value is a ternary expression using the operators "**`?`**" and "**`:`**." If the comparison condition is satisfied, it assigns the message "**`Today is Monday`**," and if it is not, the assigned value is "**`Today is not Monday`**," as shown in *Figure 7.17*:

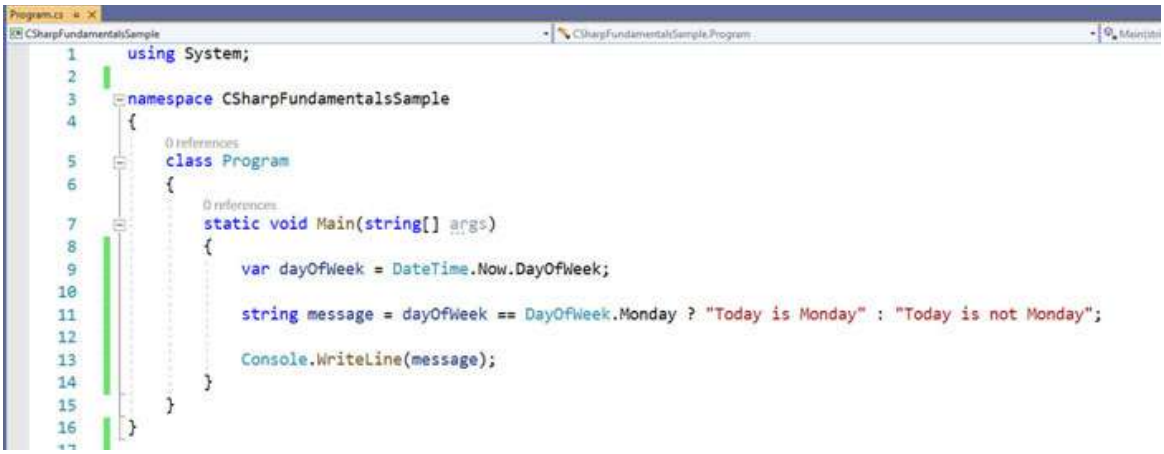


Figure 7.17: *Ternary operators in C#*

Using conditions using ternary operators is not recommended in cases where the condition is complex and contains compound statements. It would make the code hard to read. Use the "if" statement instead, as the intention of the logic operation will be explicitly exposed in future changes in the underlying part of the code.

Compound assignment operators

As with the ternary and unary operators, the bitwise type helps reduce the number of coded lines, making certain operations easier to make. Any arithmetic operator followed by the equal operator "**=**" means to the compile in C# language that the arithmetic operation before the equal operator should be applied to its variable value. *Figure 7.18* contains an example of the compound assignment operator for "**+**" (addition) and "*****"(multiplication):

```
int firstExample = 25;
firstExample += 10; // 25 + 10

int secondExample = 15;
secondExample *= 5; // 15 * 5
```

Figure 7.18: *Compound assignment operators*

If the logical implementation contains a complex operation, using compound assignment operators can complicate the algorithm's reading by developers who will maintain the code. Therefore, consider using explicit variables for complex math operations in case it contains many parts and conditions.

If statement

The C# language contains a bunch of operators for making different conditions in boolean expressions. This clearly indicates the importance of controlling the behavior of the software by implementing precise statements, even for complex conditions. To properly use the operators, you must know the “if” statement, which allows us to execute a specific part of code if a particular condition is satisfied. Otherwise, it is possible to specify an implementation that will be run in case the condition is not met. This kind of logical statement is one of the most commons operations in any programming language, and C# language has a straightforward syntax, as shown in *Figure 7.19*:

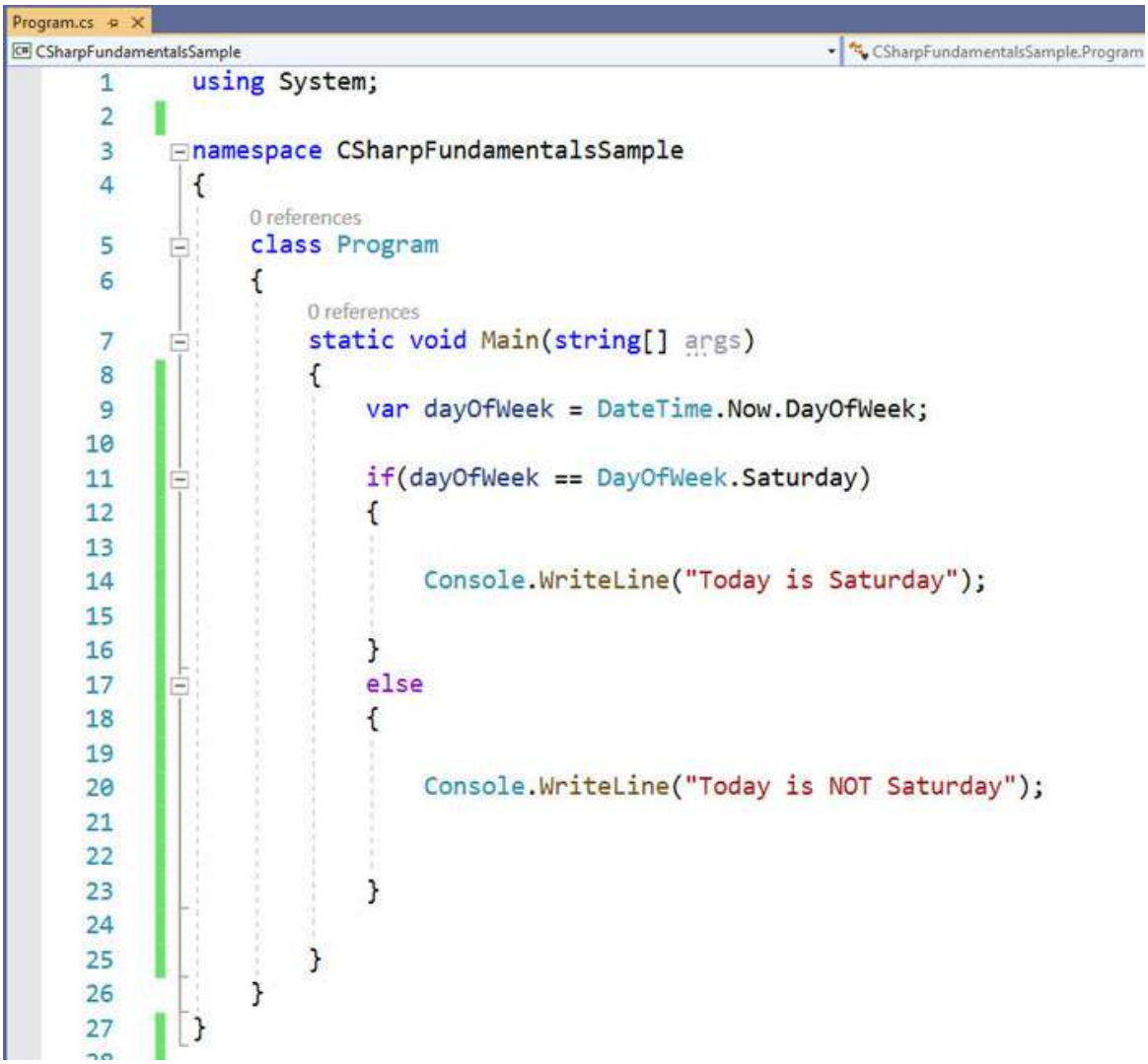


Figure 7.19: *if statement*

In this example, a specific block of code will run if the day of the week equals “Saturday,” and a different block will run if the condition has a false result. Using

it precisely like that is suitable mainly in cases when two different things should be executed depending on the condition result. It follows the pattern if-then-else, and it is one of the most used features in C#.

Suppose a logical requirement contains many sub-conditions inside an “if” or “else” statement. In that case, using other language structures, such as “switch” is recommended, which will be explained in the next topic. Depending on the complexity, many levels and sub-levels of conditions in a single routine could turn the code hard to read, and errors might be introduced while changes are made.

Switch case statement

The switch case statement is suitable to use if a logical implementation contains multiple options to choose from; thereby, only one will have its block of code executed; in other words, the option which matches with the pre-defined value for comparison. The switch case is largely used if a value should be tested by many conditions or even if it has more than two or three conditions, as shown in *Figure 7.20*:

```
9           var dayOfWeek = DateTime.Now.DayOfWeek;
10
11          switch(dayOfWeek)
12          {
13              case DayOfWeek.Monday:
14                  Console.WriteLine("Today is Monday");
15                  break;
16              case DayOfWeek.Tuesday:
17                  Console.WriteLine("Today is Tuesday");
18                  break;
19              case DayOfWeek.Wednesday:
20                  Console.WriteLine("Today is Wednesday");
21                  break;
22              case DayOfWeek.Thursday:
23                  Console.WriteLine("Today is Thursday");
24                  break;
25              case DayOfWeek.Friday:
26                  Console.WriteLine("Today is Friday");
27                  break;
28              case DayOfWeek.Saturday:
29                  Console.WriteLine("Today is Saturday");
30                  break;
31              case DayOfWeek.Sunday:
32                  Console.WriteLine("Today is Sunday");
33                  break;
34          }
```

Figure 7.20: *Switch statement*

In this example, a value (day of the week) is tested against seven options, one for each “case” in the instruction. If one of the conditions is satisfied, the program will execute the block of code specified between the underlying “case” and the “break.” It is beneficial to expose system requirements if the workflow contains multiple options and the routine that should be executed for each one is slightly different. A “switch” statement for multiple options instead of “if” is a good practice in C# programming language. If, in the example presented in the following image, the “if” statement was used, the code would look like that, which is not recommended, as shown in *Figure 7.21*:

```
var dayOfWeek = DateTime.Now.DayOfWeek;

if(dayOfWeek == DayOfWeek.Monday)
{
    Console.WriteLine("Today is Monday");
}
else
{
    if (dayOfWeek == DayOfWeek.Tuesday)
    {
        Console.WriteLine("Today is Tuesday");
    }
    else
    {
        if (dayOfWeek == DayOfWeek.Wednesday)
        {
            Console.WriteLine("Today is Wednesday");
        }
        else
        {
            if (dayOfWeek == DayOfWeek.Thursday)
            {
                Console.WriteLine("Today is Thursday");
            }
            else
            {
                if (dayOfWeek == DayOfWeek.Friday)
                {
                    Console.WriteLine("Today is MondFridayay");
                }
                else
                {
                    if (dayOfWeek == DayOfWeek.Saturday)
                    {
                        Console.WriteLine("Today is Saturday");
                    }
                    else
                    {
                        if (dayOfWeek == DayOfWeek.Sunday)
                        {
                            Console.WriteLine("Today is Sunday");
                        }

                    }
                }
            }
        }
    }
```

Figure 7.21: *Bad practice of if statement*

This implementation works the same way as the preceding example, but it contains a significantly greater number of lines and is much more complex to

read. Therefore, using the "`switch`" statement is recommended when a value needs to be tested against multiple values.

Points to remember

- The C# language contains many ways to make loop operations with similar behavior but slightly different purposes.
- A good definition of types for variables is one of the most basic and important practices in software development, as it allows the compiler to catch errors before runtime operations.

Conclusion

As seen in this chapter, the C# language provides many features to not only make arithmetic operations but to implement logical operations using loops and other distinct structures to build robust and reliable enterprise applications with the applicability of the best practices of software development used in the market by big companies in terms of strong-typed objects concept and precision in arithmetic operations.

In this chapter, you learned essential and fundamental concepts of C#, such as variables, logical statements, operators, error handling, and different type of loops and iterations. Now, you can learn more advanced coding concepts with C# and get started on the following related topics.

In the next chapter, you will have the chance to learn basic concepts of error handling and exceptions in C# language, familiarize yourself with the most common errors that can happen in a program developed using C#, and prepare for implementing solutions to prevent those errors.

Multiple choice questions

1. **Which option contains only relational operators:**

    a. ==, !=, >, <

    b. !=, >, <, if

    c. switch, for, foreach, equal

    d. None of these

2. **Which type of variable is used to store true and false values?**

    a. string

b. int
c. bool
d. decimal

3. **What is the value of the variable “isValid” in the following code?**

a. int age = 25;
b. var isValid = age > 18;
c. false
d. true
e. 25
f. 18

4. **After the following code, what is the final value of the number variable?**

a. int number = 78;
b. number++;
c. 78
d. 77
e. 80
f. 79

Answers

1. **a**
2. **c**
3. **b**
4. **d**

Questions

1. What are the available loop statements in C#?
2. What is the best alternative to implement a logical statement that can have more than three conditions?

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Error Handling and Exceptions in C#

Introduction

In this chapter, we will walk you through the fundamental concepts of error handling in C#, giving you more familiarity with the main exceptions and with ways to prevent unexpected errors that will help you understand what you need to develop robust and reliable applications using C# language and the .NET platform.

You will learn how to create and work with try-catch blocks, the primary exceptions you can find in real. Additionally, you will be able to apply error-handling strategies while implementing basic programs in C#.

Getting familiar with the fundamental concepts of exceptions and error handling in C# is essential to understand how to apply the necessary to build resilient applications in real scenarios.

Structure

In this chapter, we will discuss the following topics:

- The try-catch blocks
- Most common exceptions in C#
- Error handling strategies

Objectives

After studying this unit, you will learn the basic exceptions in C# language. You will know how to apply error-handling strategies in multiple scenarios and understand the purpose of try-catch blocks in the C# language.

The try-catch blocks

The software development process is closely related to logical implementation and interactions between software, hardware, and third-party systems, which can cause unexpected behavior in any project. Therefore, it is necessary to use statements that allow the software to handle any error properly in case of an issue. The error handling concept is not only about providing a good user experience and showing friendly messages, but it is also about taking actions properly capturing the exceptions, and addressing the correct solution for them based on what is pre-defined in the business requirements context and the following of good software architecture practices.

A technical error thrown by the application is called an **Exception** in C# language, which is not necessarily an issue caused by the application as an expected error, which means that it is perfectly possible to use a strategy to handle that situation properly. As a modern language, the C# language allows us to transfer the handling of the exception between blocks in a sequential logic using the keywords, **try**, **catch**, **finally,** and **throw**.

The ”**try**” block is used to specify the code that is responsible for running the primary operations. An error caused in the code inside that block will throw an exception which will be managed by the “catch” block, as shown in *Figure 8.1*:

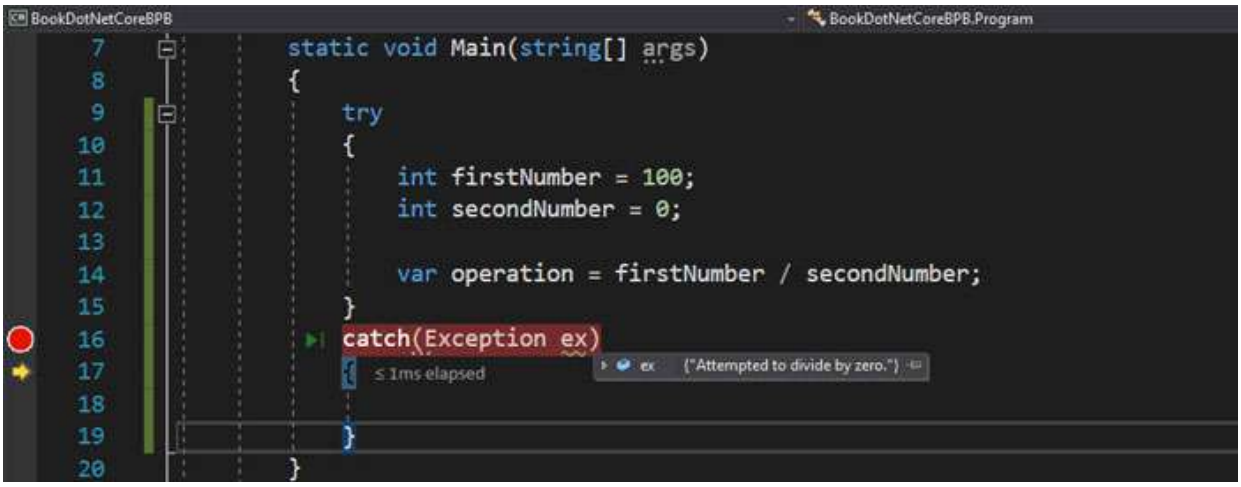


Figure 8.1: *Try block*

As it is possible to see in the preceding image, inside the try block, there is a routine to divide two numbers, and one of them clearly has a “zero” value, which throws an error considering a number cannot be divided by zero. In that case, the program throws an exception transferring the execution of the

code from the point where the exception is generated in the try block to the following exception block. Using the debug mode, it is possible to see the exception details, which is, in that case, a message indicating that there was a failed attempt in the program to divide by zero. It depends on the type of program; the execution of it would stop altogether, interrupting the whole process and, in specific cases, being necessary to restart the application. Because of this inconvenience, it is essential to use the “try” and catch blocks to handle any exception that can be thrown by the application and handle that properly, showing the user a friendly message regarding the error or recording the exception in a log or notification system.

The `catch` block is executed every time an exception occurs in the previous try block, and you must use it to handle the exception by sub-type or generically, depending on the scenario you are facing. Different types of exceptions can be caused by only one try block, and it is possible to combine multiple exceptions block in case of a specific operation needs to run for each type individually, as shown in *Figure 8*.2:

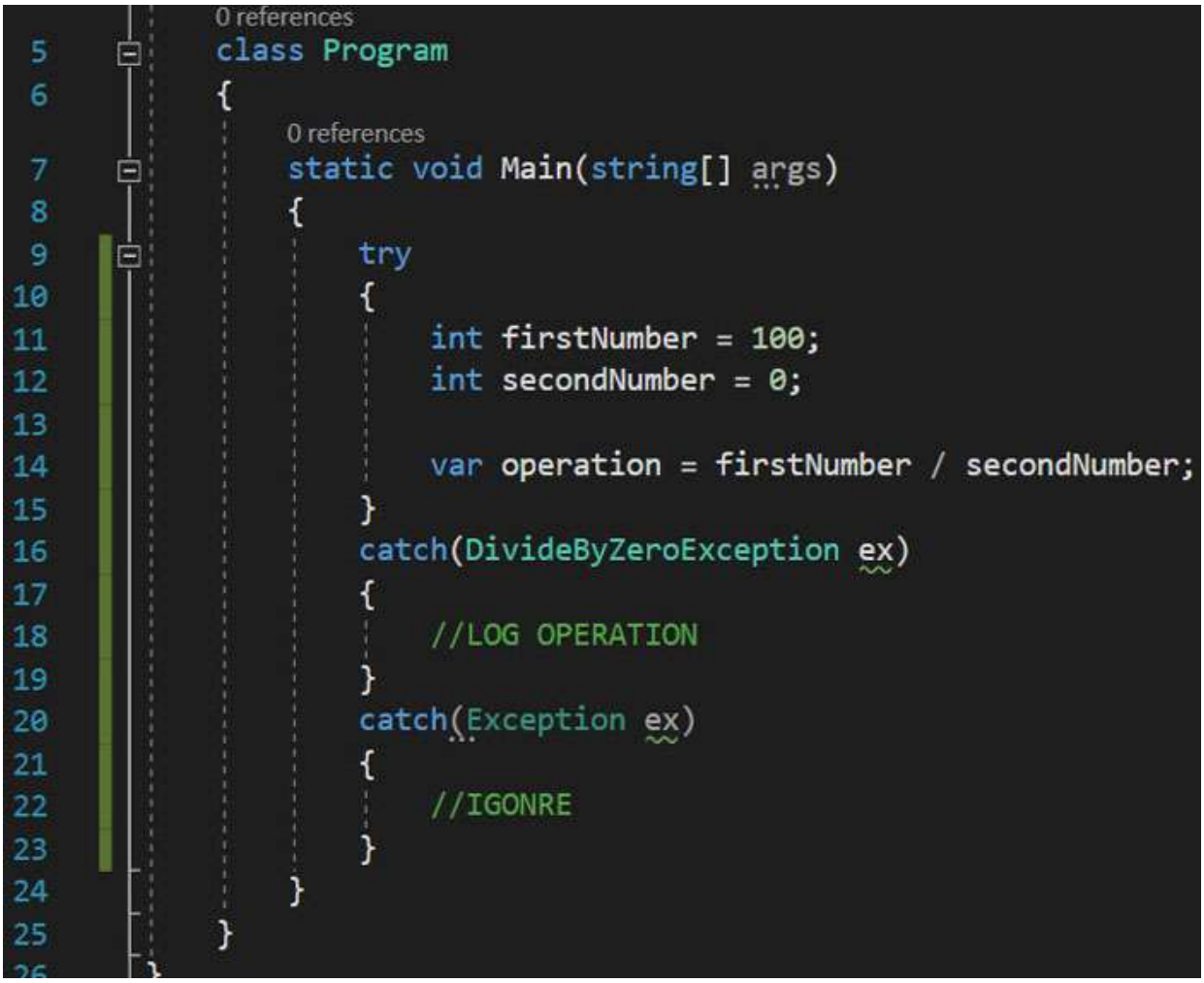


Figure 8.2: *Multiple exceptions*

In this example, there is, in the first place, a catch block checking for the occurrence of a specific exception regarding an error in the attempt to divide a number by zero, and after that, a more generic exception. Multiple exception verifications help take distinct actions and more accurate management against any error in the implementation.

Additionally, the C# language allows using of **exception filters**, which represents a more detailed verification of the exception properties as a further condition to execute the catch block or not. That resource is handy in integrations with third-party systems that usually return a range of error numbers as exceptions and allows us to implement a better error-handling strategy in many distinct scenarios. To use the exception filter feature, you must use the "**when**" keyword combined with the exception block, as shown in *Figure 8.3*:

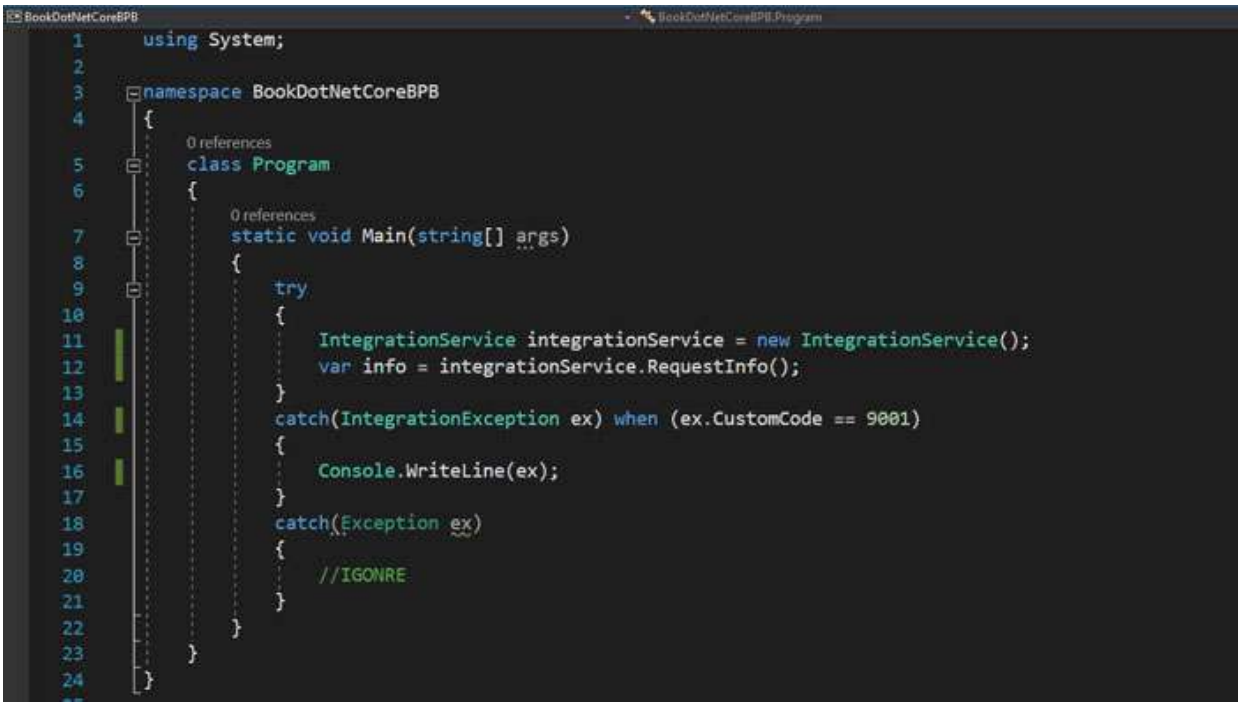


Figure 8.3: *Exception filters*

In this example, line 14 verifies if the exception code returned by the integration service equals 9001, and the underlying catch block will be executed only under this condition. This approach is primarily used in HTTP request verifications to check the status code returned by the requests. Usually, a different action is taken for each type of code, and using the "**`when`**" keyword is helpful to make a more readable code for error handling.

An extra block called "**`finally`**" can be used combined with the try and exception blocks in the cases when a specific routine needs to be executed with the previous routine is successfully executed or not, which means that the "**`finally`**" block will always run after the routine presented in the try block and the routine specified in all the exception blocks, as shown in *Figure 8.4*:

```
using System;

namespace BookDotNetCoreBPB
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            IntegrationService integrationService = new IntegrationService();

            try
            {

                var info = integrationService.RequestInfo();
            }
            catch(IntegrationException ex) when (ex.CustomCode == 9001)
            {
                Console.WriteLine(ex);
            }
            finally
            {
                integrationService.CloseConnection();
            }
        }
    }
}
```

Figure 8.4: *Exception filters*

In this case, the sample routine will consistently execute the close connection method regarding the integration service. A similar routine can be used in all cases where a routine needs to be executed after attempting to make any operations in the try-and-catch blocks.

Most common exceptions in C#

The C# language natively provides a bunch of types of exceptions. Understanding the most common in C# and .NET applications is essential to handle the errors accordingly in an application in case of some exceptions. Additionally, the knowledge of common errors allows us to spare time to find the cause of the issues and probable solutions for them once a system can present unexpected behavior. This situation would require code changes to use more specific exception verifications.

All the exceptions in C# language inherit from the Exception class, and it allows us to create our own exceptions, despite in most cases, the pre-defined exceptions are usually enough to handle the errors correctly in a

standard application. The following list represents the most common exceptions that can be found in any applications based on the C# language:

- **OutOfMemoryException**: This exception is thrown when the application tries to use more memory than is available in the system for the application precisely. It is common to find this error when the application handles large files or has a wrong implementation regarding the use of objects in the memory.
- **SqlClientException**: You will find this exception in all the cases where the application is using SQL Server or other database providers and can be caused by problems with connection or any attempt to make operations in the database.
- **StackOverflowException**: In runtime, the .NET application can identify if a specific routine contains an infinite recursive implementation, which means that the same routine can never stop because a function cannot call itself in unstoppable looping.
- **WebException**: In modern applications, it is easy to find integrations with external resources that make external calls using APIs. It requires making HTTP requests, and any problem regarding the communication with the external service may generate a WebException.
- **NullReferenceException**: This is one of the most common exceptions found in C# applications once the applications are built using object-oriented programming, and it is not possible to call any property or method of an object if the object is null itself. Considering that situation might happen in many different situations, the development of any application in the C# language requires null checks to avoid unexpected behavior in runtime, mainly in production environments.

The C# language allows us to create our own custom exceptions, which is particularly useful in cases where it is necessary to separate errors in the custom implementation from the native exceptions thrown by the .NET platform, as shown in *Figure 8.5*:

Figure 8.5: *Exception filters*

Once the custom exception class inherits from the standard Exception class used in the .NET platform, it is possible to verify the custom exception in catch blocks to implement a more accurate error handling across the application.

Error handling strategy options

The .NET Core platform introduced relevant improvements regarding error handling for Asp.Net Core applications compared to previous versions of Asp.Net applications, enabling an error handling strategy globally without affecting the entire application with different conditions and verifications.

The error handling middleware presented on Asp.Net Core will catch all the exceptions thrown by the applications, and in cases where the exception is not being handled, the application will redirect to a configured custom endpoint. Additionally, it is possible to redirect to different pages or endpoints in production and development environments, as it is not recommended to show system exceptions in environments used by final users once it can cause security to expose, revealing details of the implementation such as the database type that is being used and other technical, sensitive data used by the application.

To use a custom exception page in the development environment, you must enable it on the startup class in the Asp.Net Core application, as shown in *Figure 8.6*:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");

    }
```

Figure 8.6: *Development error page*

In the development environment, the Asp.Net Core application will redirect to an error page showing more details of the exception comparison to the one used in other environments. The **`UseDeveloperExceptionPage`** method does not have any parameters, and it is not possible in that case to redirect to a custom page in the application. Additionally, it is possible to use the standard status code error for production non-development environment using the following directive:

```
0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseStatusCodePages();


        app.UseExceptionHandler("/Home/Error");

    }
```

Figure 8.7: *Status code page*

All the configurations regarding error handling in the Asp.Net Core application aim to redirect only the unhandled exceptions to a specific

endpoint. It is common to have a scenario where during development time, it is not possible to identify all the exceptions or errors the application can have. Therefore, it is essential to manage all unexpected behavior in an enterprise application properly.

Conclusion

As seen in this chapter, the C# language provides many features to handle appropriately all the errors and exceptions that can exist in enterprise applications, being possible to define strategies for error handling to build robust and reliable applications in production environments, following the best practices of security and user experience in the cases where the application has unexpected behavior.

In this chapter, you learned fundamental concepts of exceptions in C#, such as try, catch, and finally, blocks. Additionally, you have the opportunity to understand how to handle errors in Asp.Net Core applications and learn how to create your custom exception in C# language.

In the next chapter, you will have the chance to learn basic concepts of **Language-Integrated Query** (**LINQ**) in C# language and get yourself familiar with the most common use of LINQ combined with operations in List objects.

Points to remember

- The C# language provides many native and standard exceptions that can be used to handle enterprise application errors.
- It is possible to use the try-and-catch blocks to manage the errors in .NET applications.
- The Asp.Net Core contains relevant improvements regarding error handling, which allows us to spend less time implementing custom error-handling strategies in real-case scenarios.

Multiple choice questions

1. **Which type of exception is thrown by .NET applications when a routine tries to use a method or property of a null object?**

a. Try
b. OutOfRangeException
c. NullReferenceException
d. None of these

2. **Which block of code will run in case of handled errors in a .NET application?**

a. null
b. finally
c. try
d. catch

3. **In which class can we specify a global error handling in Asp.Net Core applications?**

a. Program class
b. Start class
c. Exception class
d. None of these

Answers

1. **c**
2. **d**
3. **b**

Questions

1. Explain how to create custom exceptions in C# language.
2. Why is it not recommended to show the technical details of an error in production environments?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Using and Understanding LINQ

Introduction

In this chapter, we will walk you through the basic fundamental concepts of **Language-Integrated Query** (**LINQ**) in C#, thus allowing you to learn and practice the most common operations that can be made by manipulating IEnumerable objects in the .NET platform.

You will learn how to create expressions, filters, and primary operations for manipulating data in C# language based on Lists and Enumerable objects, such as data returned from external resources, APIs, and databases.

Getting familiar with LINQ's basic and essential concepts is crucial to understanding how to apply the necessary to build applications in real scenarios using expressive, readable, and performant code statements in data manipulation.

Structure

In this chapter, we will discuss the following topics:

- LINQ fundamentals
- Query expressions

Objectives

After studying this unit, you should be able to know the basic operations using LINQ and apply LINQ statements in real scenarios. You will also be able to identify possible improvements in legacy projects regarding LINQ operations.

LINQ fundamentals

The software development process has evolved in the last few years to use the most recommended practices in terms of abstraction, reusability, and readable code produced by developers. In many scenarios, a system consisting of data manipulation interacting with a database and other external resources is extremely necessary to create an abstraction in any application to simplify not only the access to databases but also operations made on arrays and standard collections in the C# language. Over the later years, professional specialization became essential to reach the full potential of knowledge applicability in many technologies, including C# language and .NET platform. Therefore, it is pretty common to find situations where a .NET developer builds an entire system without writing any database query, coding all the routines using only the C# language and using a specific framework to abstract the database operations, such as Entity Framework Core, which will be explained with details later in this book.

Usually, the data manipulation in C# language requires the use of **Language Integrated Query** (**LINQ**), which simplifies the standard operations in collections and allows us to keep consistent code across the application once all the operations can be made only using the C# language. The LINQ possibilities have been improved in the last releases of .NET Core in terms of performance and parallel operations, and it was consolidated among .NET developers. They can keep the focus on learning the C# language deeply, making powerful operations in terms of data manipulation at a code level.

LINQ is the technology that allows us to make query operations in objects using only C# language, possibly manipulating data from a vast amount of distinct data sources, such as XML, SQL Server, and many others, including any object that implements the IEnumerable interface. The logic behind LINQ operations is quite similar to the concept of SQL queries, including filters, order and group operations, inner joins, and much more. Therefore, if you are already familiar with SQL queries, adapting to LINQ operations will be much easier considering their clear existing correlation. However, the knowledge of SQL queries is not mandatory to learn LINQ operations, and you will have the opportunity to practice from the most basic to advanced operations using LINQ during this chapter.

Query expressions

To get started with LINQ queries in this section, a hypothetical scenario that includes a sort of class must be defined to have a set of models to apply the most common LINQ operations. First of all, after creating a Console application using the most recent .NET version and C# language, create a class called “ProductType,” like the structure shown in *Figure 9.1*:

```
11 references
5   public class ProductType
6   {
        6 references
7       public int Id { get; set; }
        3 references
8       public string Name { get; set; }
9
10  }
11
        9 references
12  public class Product
13  {
        4 references
14      public int Id { get; set; }
        5 references
15      public string Title { get; set; }
        3 references
16      public ProductType ProductType { get; set; }
        4 references
17      public decimal Price { get; set; }
        4 references
18      public int StorageQuantity { get; set; }
        4 references
19      public bool Active { get; set; }
20
        2 references
21      public List<Purchase> Purchases { get; set; }
22  }
```

Figure 9.1: *Product Type and Product classes*

Additionally, in this example, it is necessary to create two extra classes regarding customer and purchase information, with the purpose to apply more variations of LINQ operations and to have more query expression options. Both of the classes have the following structure as shown in *Figure 9.2*:

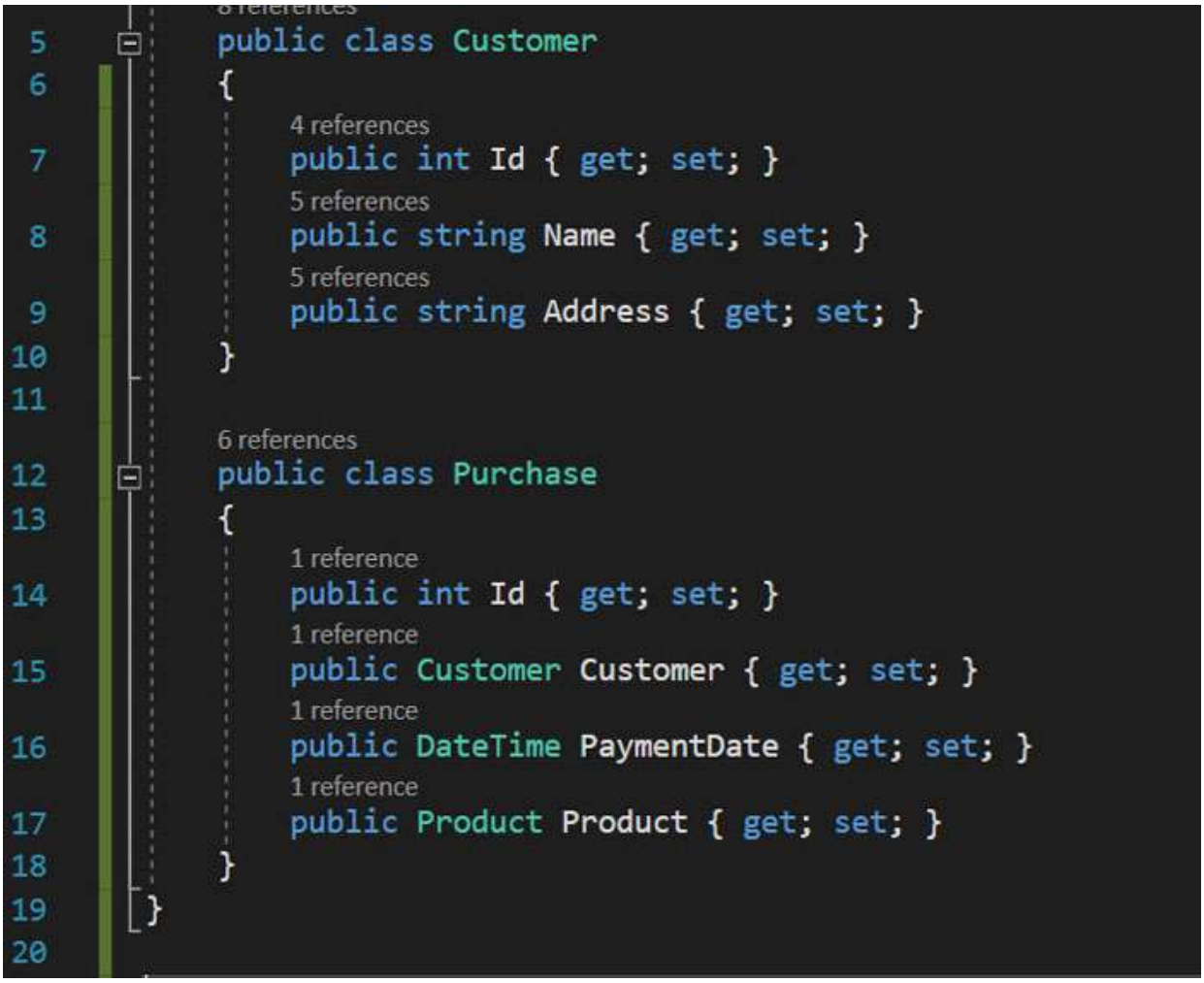


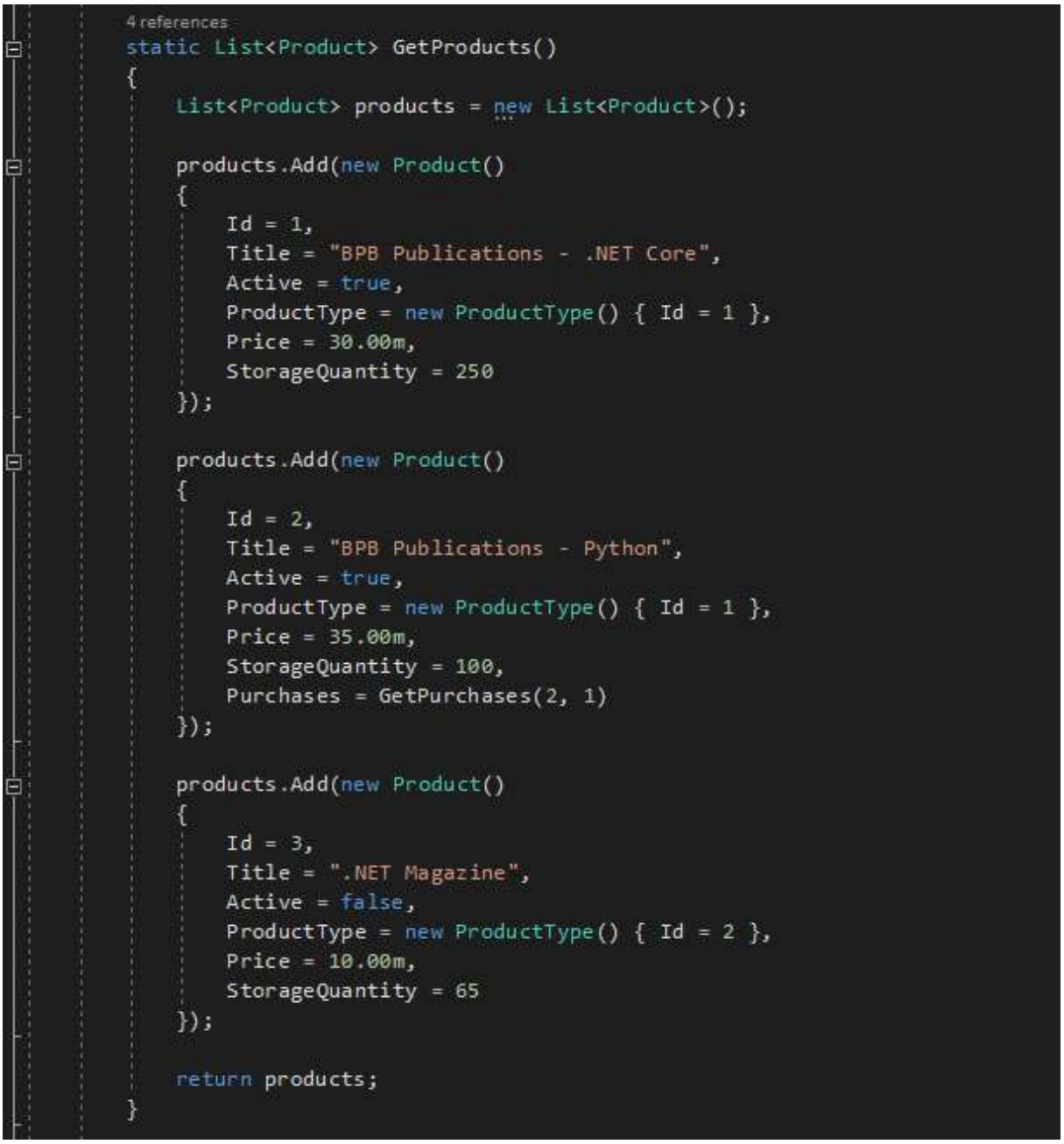
Figure 9.2: *Customer and Purchase classes*

As there is already a set of classes for making the distinct type of LINQ queries, the next step is to generate mock data that would represent data that comes from a database, similar to what would be found in a real scenario. The samples in this section are close to situations where an Object-Relational Mapper (ORM) is used to bring information from the database and parse the result of SQL queries to C# objects. As a final step for having the environment for LINQ statements, you must create a sort of data regarding all the entities (product, product type, customer, and purchase), as shown in *Figure 9.3*, starting from the Product Type model:

```
0 references
static List<ProductType> GetProductTypes()
{
    List<ProductType> productTypes = new List<ProductType>();

    productTypes.Add(new ProductType() { Id = 1, Name = "Book" });

    productTypes.Add(new ProductType() { Id = 2, Name = "Magazine" });

    productTypes.Add(new ProductType() { Id = 3, Name = "Laptop" });

    return productTypes;
}
```

Figure 9.3: *Get product types method*

This context has three distinct product types, which will be associated with the Product entity. You can place the previous static method in the Program for test **`purposes.cs`** file (Console application) or create it in a separate file for your convenience. As a sort of product type was already created, the next step is to create a sort of product, as shown in *Figure 9.4*:



```
4 references
static List<Product> GetProducts()
{
    List<Product> products = new List<Product>();

    products.Add(new Product()
    {
        Id = 1,
        Title = "BPB Publications - .NET Core",
        Active = true,
        ProductType = new ProductType() { Id = 1 },
        Price = 30.00m,
        StorageQuantity = 250
    });

    products.Add(new Product()
    {
        Id = 2,
        Title = "BPB Publications - Python",
        Active = true,
        ProductType = new ProductType() { Id = 1 },
        Price = 35.00m,
        StorageQuantity = 100,
        Purchases = GetPurchases(2, 1)
    });

    products.Add(new Product()
    {
        Id = 3,
        Title = ".NET Magazine",
        Active = false,
        ProductType = new ProductType() { Id = 2 },
        Price = 10.00m,
        StorageQuantity = 65
    });

    return products;
}
```

Figure 9.4: *Get product method*

As seen in the preceding image, the method creates a list of three products, each with specific property values. It is essential to have collections with diverse configurations and data variations to see relevant results in LINQ queries. Finally, the last step for the mock data setup is the specification of a method responsible for generating purchases, which is referred to in *Figure 9.4* in the second collection item. The method has the structure as shown in *Figure 9.5*:

```
1 reference
static List<Purchase> GetPurchases(int productId, int customerId)
{
    List<Purchase> purchases = new List<Purchase>();

    for (int i = 1; i > 5; i++)
    {
        purchases.Add(new Purchase
        {
            Id = i,
            PaymentDate = DateTime.UtcNow.AddDays(-i),
            Customer = new Customer() { Id = customerId },
            Product = new Product() { Id = productId }
        });
    }

    return purchases;
}
```

Figure 9.5: *Get product method*

As the setup is ready, it is possible to start your journey with LINQ expressions, starting from the extension method and the most common filter operations. The query structure is quite similar to the logic behind SQL queries. The standard C# operators, which were already explained previously in this book, must be used to specify conditions, from the simple to the most complex filters. Considering the mock data that is being used in the samples of this section, if, in a hypothetical scenario, a specific routine in the system has the purpose of filtering all the products that the price is smaller than 20, the LINQ expression would be like the representation as shown in *Figure 9.6*:

```
var products = GetProducts();

var productFilter = products.Where(x => x.Price < 20).ToList();

foreach(var product in productFilter)
{
    Console.WriteLine($"Product: {product.Title}. Price: {product.Price} ");
}

Console.ReadLine();
```

Figure 9.6: *Get product method*

As shown in *Figure 9.6*, the extension method “Where” is being used and receiving a delegate function containing boolean expressions as a parameter is seen in the “**`productFilter`**” variable assignment. The query is returning all the products that satisfy the condition of having a price smaller than 20. After making the filter, the extension method “**`ToList`**” is used to convert the result from **`IQueryable`** to **`List<Product>`** type. A loop shows the result (product title and price) in the Console. As the product collection has only one product with a price smaller than 20, the product filter list contains only one item, as shown in *Figure 9.7*:

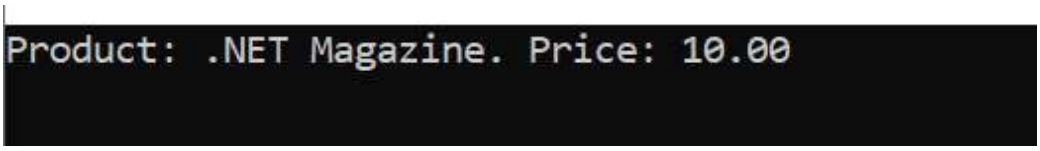


Figure 9.7: *Product filter result*

If the LINQ query does not return any item, the result would be a null collection after converting to the List type. Because of that, it is essential to handle the filter results correctly after the LINQ operations to avoid exceptions like “Null Reference Exception”. Furthermore, in cases, where the LINQ expressions are being used with the Entity Framework or other ORM, it is essential to use only mapped properties because the ORM tries to convert the LINQ expressions to SQL queries, and the properties used in the statement should match to the columns in the database table.

Continuing the example using the product list, considering the LINQ expression can be used as an extension method, it is possible to combine multiple statements, such as the order of the result after applying a specific filter. The following *Figure 9.8* represents an ascending order of the product list, using the title property:

```
var products = GetProducts();

var orderedList = products.OrderBy(x=> x.Title).ToList();

foreach (var product in orderedList)
{
    Console.WriteLine($"Product: {product.Title}");
}

Console.ReadLine();
```

Figure 9.8: *Order by the statement*

All the extension methods expect to receive a delegate **`Func`** type; therefore, even for developers who don't have experience with LINQ expressions, it is pretty easy to get familiar with it, as all the methods follow the same pattern. Additionally, the expressions can follow the same order as a SQL query; therefore, a SELECT statement followed by the filter is highlighted as shown in *Figure 9.9*:

```
var products = GetProducts();

var titleList = products.Select(x=> x.Title).Where(x=> x.Contains("Magazine")).ToList();

foreach (var title in titleList)
{
    Console.WriteLine($"Product: {titleList}");
}

Console.ReadLine();
}
```

Figure 9.9: *Select statement*

As the product title property is of a string type, the list returned by the LINQ operations, followed by the **`ToList`** method, will return a collection of string objects instead of the whole product object. This approach represents good practice, mainly when data comes from databases. It increases the performance and consumes much less server memory. In this case, it is necessary to bring only one object from the list being the criteria of the first or last item on the list, and it is possible to use the FirstOrDefault or LastOrDefault methods, as shown in *Figure 9.10*:

```
{
    var customers = GetCustomers();

    var firstCustomer = customers.FirstOrDefault();
    var lastCustomer = customers.LastOrDefault();

    Console.WriteLine($"First customer: {firstCustomer.Name}");
    Console.WriteLine($"Last customer: {lastCustomer.Name}");


    Console.ReadLine();
```

Figure 9.10: *FirstOrDefault and LastOrDefault methods*

In that case, the result in both cases is a product object in case there is at least one item in the product list. The query returns a null product object if the product contains no items. An alternative to that approach is to use the First and Last extensions methods. However, they would return a system exception if the product collection does not have any product. Therefore, it is recommended to use FirstOrDefault and LastOrDefault if the length of the collection is uncertain.

The conditional statements in the extension methods can have multiple operations, combining many conditions. In this scenario, mainly when the query contains more than three conditions, keeping the indentation and the code as readable as possible, it is imperative for an excellent visual interpretation. All the conditional operators can be applied in this context, as shown in the following example:

```
var products = GetProducts();

var productFilter = products

    .Where(x => (x.Active == true && x.StorageQuantity > 50) ||
                x.Purchases.Count > 1);

foreach (var product in productFilter)
{
    Console.WriteLine($"Product: {product.Title}. Price: {product.Price} ");
}
```

Figure 9.11: *Combined conditional statements*

The current section of this book aims to demonstrate the most common LINQ extension methods used in .NET Core applications, and more

advanced statements are further explained in *Chapter 18, 'Interacting with Databases Using the Entity Framework Core'*, where the LINQ expressions are combined with the Entity Framework. However, before looking at more advanced LINQ expressions, it is essential to get familiar with the most simple statements to understand the fundamental concepts behind delegate **Func** type, lambda expressions, and conditional operators.

The LINQ expressions contain many extension methods available that can be used to execute advanced filters with simplicity in terms of syntax and good performance, even when a collection gets relevant items to count. In specific scenarios, it is better to use the Boolean return to check the occurrence of items based on filters instead of bringing the full objects and checking if the list is null or not, as shown in *Figure 9.12*:

```
var customers = GetCustomers();

bool hasCustomer = customers.Any(x => x.Address.Contains("New York"));

if(hasCustomer)
{
    Console.WriteLine("It has a customer with the New York keyword in the address");
}
```

Figure 9.12: *The use of Any statement*

In that case, the LINQ query will return true or false depending on the occurrence of an object in the list that satisfies the condition. It represents a better alternative to the Count method, mainly in cases when the Entity Framework is being used. Remember that not only delegate **Func** expressions can be passed as parameters in specific LINQ extension methods, but also other types of objects such as other lists, Booleans, and much more. The following example represents the Except and Intersect methods, both receiving a list as a parameter. The first one excludes from the result all the items presented in the second list, and the second example demonstrates how to combine two lists using the Intersect method, as shown in *Figure 9.13*:

```
customers = Customer.GetCustomers();
var customersNewYork = customers.Where(x => x.Address.Contains("New York")).ToList();

var allCustomerExceptNewYork = customers.Except(customersNewYork);

Console.WriteLine("Except result");
Console.WriteLine(String.Join(", ", allCustomerExceptNewYork.Select(x=> x.Address)));


List<int> firstList = new List<int> { 1, 2, 3,4, 5 };
List<int> secondList = new List<int>() { 1,2,8,9,10 };

List<int> intersectList = firstList.Intersect(secondList).ToList();

Console.WriteLine("Intersected List");
Console.WriteLine(String.Join(",",intersectList));
```

Figure 9.13: *Except and Intersect methods*

Apart from all the methods demonstrated during this section, the C# language provides many other options, such as Zip, Group By, Inner Joins, Distinct, Append, Find, Aggregate, and many others, which are shown in detail in *Chapter 18, 'Interacting with Databases using the Entity Framework Core'* of this book regarding the Entity Framework. As the manipulation of collections in C# language is one of the most common operations in many systems, being hugely used combined with ORMs, the current section has the purpose of showing the most basic LINQ expressions, which will be helpful to get yourself familiar with it for the upcoming chapters of this book.

Points to remember

- The C# language provides many LINQ extension methods, most accepting lambda expressions.
- LINQ expression is commonly used combined with the Entity Framework and other ORMs.
- The performance of operations in a list of objects is directly affected by the type of extension method used.

Conclusion

As seen in this chapter, the C# language provides many features to properly handle all the necessary operations on collections and other types of objects, which implements the IEnumerable interface using LINQ operations such as

filter, order, intersection, and many others. The use of LINQ allows us to abstract SQL queries as well if an ORM is used in the system. In that case, the Entity Framework translates the LINQ statement to the SQL query.

In this chapter, you learned essential and fundamental concepts of LINQ in the C# language, such as filters, combined conditional statements, and intersections. Additionally, you had the opportunity to understand how to learn details on LINQ that will allow you to understand the Entity Framework Core deeply further in this book.

In the next chapter, you will learn basic concepts of Unit Tests in C# language and familiarize yourself with the concept of TDD (Test-Driven Development) and Xunit.

Multiple choice questions

1. **Which method in the list given below does not belong to LINQ extension methods in the C# language?**

    a. Any()

    b. Where()

    c. FirstOrDefault()

    d. Raw()

2. **Which extension method can be used to combine the items of two lists:**

    a. Except()

    b. Intersect()

    c. OrderBy()

    d. Zip()

3. **What LINQ method allows us to return a boolean value (true or false) based on the filter result?**

    ToArray()

    a. ToList()

    b. Equal()

    c. Any()

Answers

1. **d**
2. **b**
3. **d**

Questions

1. Explain the main purpose of LINQ extension methods.
2. Using the knowledge you have gained in this chapter, create variations of the collections used in this chapter and create at least one filter for each entity (product, product type, customer, and purchases).

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

Unit Tests

Introduction

In this chapter, you will learn the fundamental concepts of unit tests using C# language, and you will also have the opportunity to apply the new knowledge in practical examples, use unit tests in real scenarios, and improve the existing ones in legacy projects.

You will learn how to create basic unit tests in C# projects using the xUnit tool, a robust test suite for .NET projects. You will get familiar with **Test-Driven Development** (**TDD**) concepts and understand how to create and run unit tests effectively.

Getting familiar with unit tests for .NET projects is fundamental knowledge to increase the quality of the projects you are working on and reduce the risk of issues in production environments, giving you a chance to make changes safely, even in legacy projects.

Structure

In this chapter, we will discuss the following topics:

- Unit test concept
- Unit test samples with the xUnit tool
- **Test-Driven Development** (**TDD**) concept

Objectives

After studying this unit, you should be able to code unit tests in legacy and new .NET projects, use the xUnit tool to write basic unit tests and understand how to apply unit tests in real scenarios, and identify possible improvements that can be made in legacy code to have classes more testable.

Unit Test Concept

Testing is one of the essential phases in a software development life cycle, representing, in many scenarios, the key point to reduce the project costs and provide quality and reliability to the software. The art of making tests involves professionalism, techniques, and ample experience in the software development process and, sometimes, it requires strong knowledge of business requirements related to the software, even in small projects, which can easily have hundreds or thousands of tests to cover all the requirements and to make sure every routine is working as expected. All the software-related activities involve manual processes at a certain point made by a professional specialized in test counting with a large project experience and its functionality. However, manual tests may have mistakes in their execution, considering it is not always possible for a professional to remember all the requirements, possibilities, and workflow regarding certain functionalities. Additionally, a functionality could contain a specific implementation that would require repeated tests to ensure that it is working as expected.

For the given reasons, implementing automated tests is highly recommended because a good level of automation gives the test process consistency, reliability, integrity, and a trustable result regarding regression tests. Having said that the unit tests consist of writing automated to test a single unit of the code with precise expected results. In the lowest levels of automated tests, once a single test is responsible for checking whether a tiny expected behavior of a function, method, or class is being correctly satisfied and implemented, meeting all the defined requirements for this small part of the software.

A good approach in unit tests is strongly related to the Single Responsibility Principle (SRP) of the SOLID principles; once this specific principle helps us to achieve the goal of having testable classes. Considering a unit test is responsible for testing a specific part of code, if the same code contains multiple responsibilities or its requirements are unclear, it would get harder to define and write unit tests for the underlying code. Therefore, applying a refactor in a legacy project might be necessary for many scenarios with testable classes.

After the summary of the unit tests concept, you will find practical examples of tests using the xUnit tool for .NET projects in the following sections of

this chapter. To apply and reproduce the examples in the next chapter, it is essential to have small familiarity with object-oriented programming about what you can find helpful information in the previous chapter of this book.

xUnit tool for .NET

The xUnit is a primarily used testing tool for .NET projects, including .NET Framework, .NET Core, and .NET Standard, being a consolidated library to write a sort of distinct tests not only for C# language but also for other extra languages supported by the .NET platform. The tool gives us the possibility to extend its native functionality and customize it according to our specific scenario. The constant updates and support given by the community are one of the important benefits of the xUnit tool once it is a free and open-source project. Therefore, this tool represents a good alternative for projects of any size and is perfectly suitable for almost all scenarios that can be found in standard and common scenarios. To use the xUnit tool, it is necessary to install the underlying package in a Test Project, as shown in *Figure 10.1*:

Figure 10.1: *xUnit packages*

For demonstration purposes, a hypothetical scenario is defined during this section, counting on a number of classes and methods that allow us to apply the basic features provided by the xUnit tool and to cover the most basic concepts of unit tests. The scenario consists of an online store application that contains specific requirements depending on the type of product, as shown in the following list:

- The system should not allow the specification of address delivery information in the case of online products.
- All the books in the online store must provide a twenty percent discount.

- All the customers must have a thirty percent discount on their anniversary.
- The discount is never cumulative if there is a conflict between the requirements. The system should provide the greater one in that case.

Considering these requirements and the given scenario, the project would have, in the hypothetical and specific scenario, simplified classes and methods starting with the customer class, as shown in *Figure 10.2*:

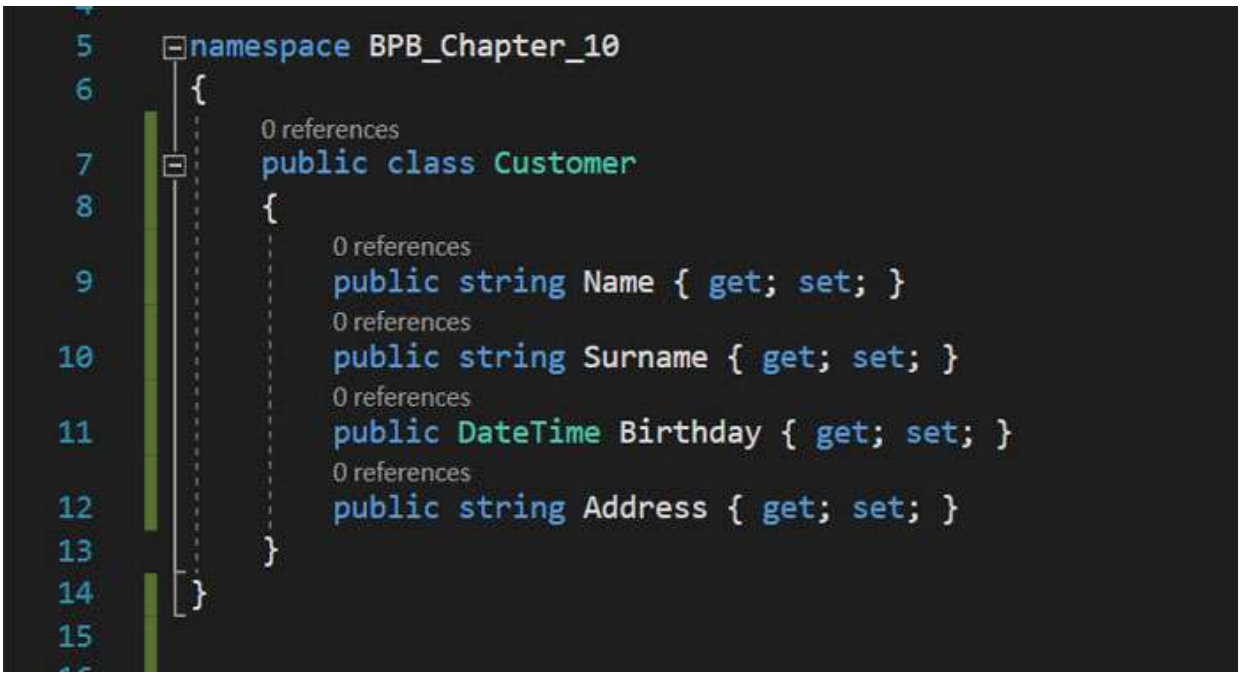


Figure 10.2: *Customer class*

The next class is regarding the product class, including an enum for product type and a basic method to calculate discount following the underlying scenarios, as presented in *Figure 10.3*:

```
using System;

namespace BPB_Chapter_10
{
    3 references
    public class Product
    {
        1 reference | 1/1 passing
        public int Id { get; set; }
        1 reference | 1/1 passing
        public string Title { get; set; }
        4 references | 1/1 passing
        public decimal Price { get; set; }
        2 references | 1/1 passing
        public ProductType ProductType { get; set; }
        2 references | 1/1 passing
        public bool IsOnlineProduct { get; set; }

        1 reference | 1/1 passing
        public decimal ApplyDiscount(Customer customer)
        {
            if (customer.Birthday.Day == DateTime.Now.Day &&
               customer.Birthday.Month == DateTime.Now.Month)
            {
                return this.Price * 0.70m;
            }

            if (ProductType == ProductType.Book)
            {
                return this.Price * 0.80m;
            }

            return this.Price;
        }
    }

    3 references
    public enum ProductType
    {
        Book = 1,
        Laptop = 2,
        Magazine = 3
    }
}
```

Figure 10.3: *Product class*

In that case, the product class has a method responsible for calculating the discount based on the requirements, and if none of the conditions is satisfied, the method returns the original product value. The last class that needs to be created in the scenario is the Order class, where the address validation is

placed to follow the requirements to avoid shipping address for online products, as shown in *Figure 10.4*:

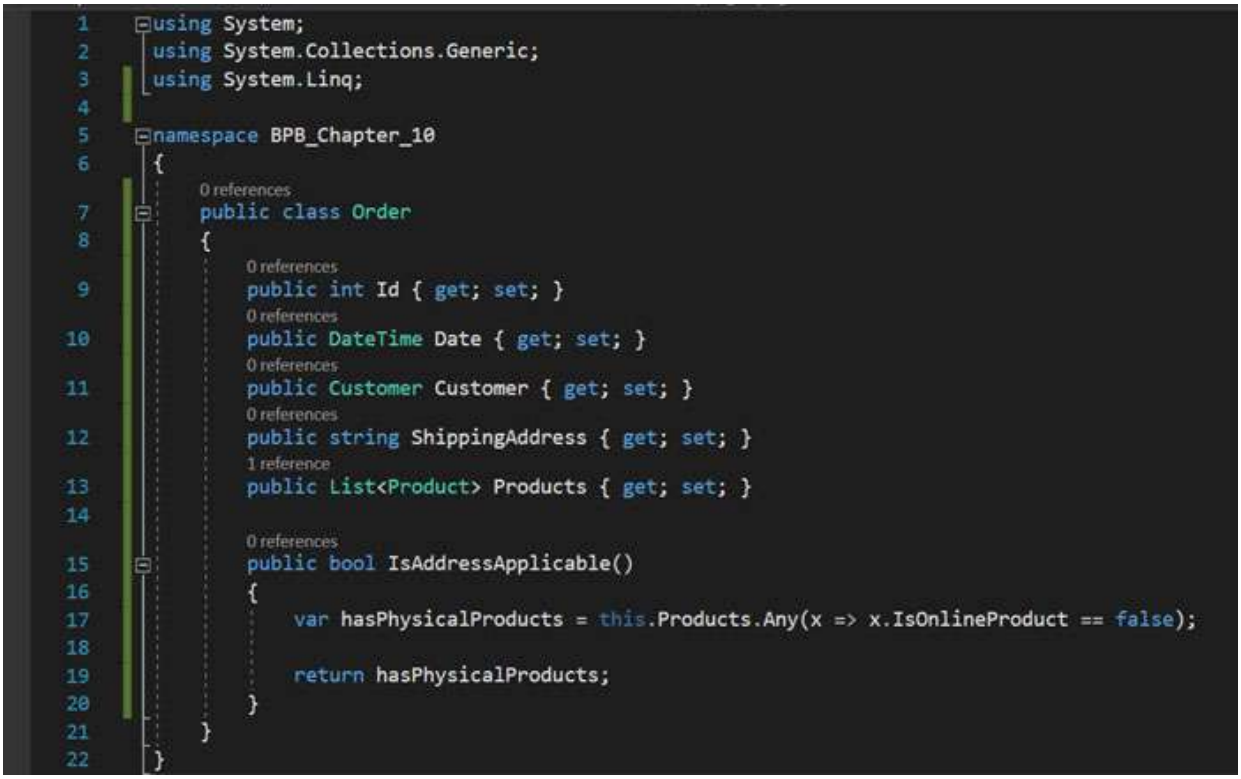


Figure 10.4: *Order class*

The class has a method that returns a boolean value if there is at least one physical product as part of the order, meeting the requirement in the hypothetical scenario. Although the three classes are simplified, there are a lot of test cases that are applicable to check each one of the requirements. To be testable, an individual class or method must have a clear and unique purpose and needs to allow the simulation of its behavior accordingly. If for some reason, a method depends on external resources to run, such as databases and APIs, the best approach in the case would be the implementation of integration tests instead of unit tests, as the part of the code that needs to be tested clearly has more than one responsibility in that case. In many scenarios, it might represent an indication that the method should be split into other smaller methods, reducing the dependency between them and getting easier to write unit tests for the individual methods separately.

First, we must create new test classes responsible for running the tests for each business class (customer, order, and product) to organize the tests into

recognizable and meaningful-related parts. The first test class is the **`ProductTest`** class, as shown in *Figure 10.5*:

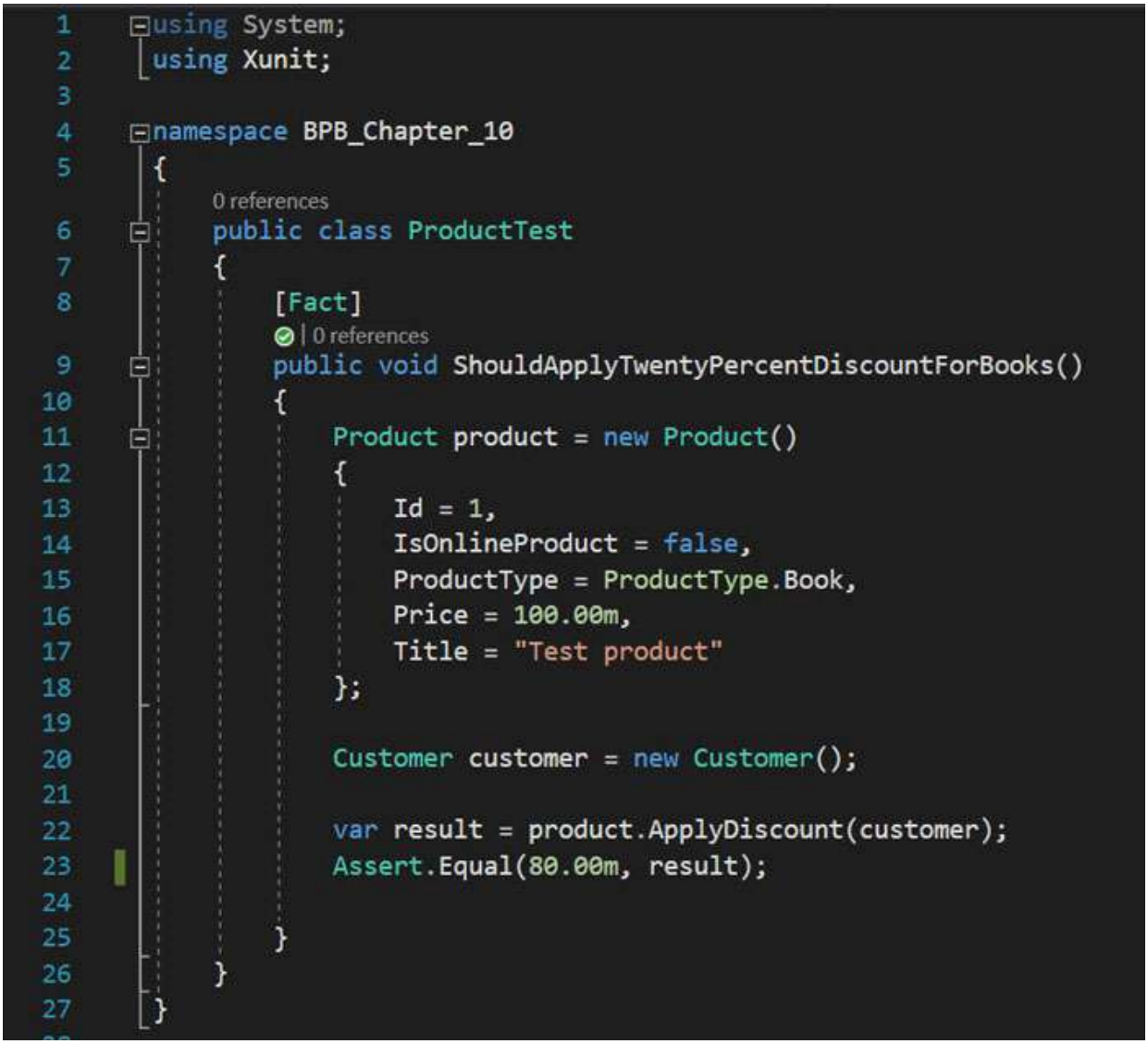


Figure 10.5: *Product test class*

The xUnit tool requires using specific annotations associated with the test methods, such as “Fact,” “Theory,” and others. In that case, they will be recognized as test classes. In the preceding example, the method is responsible for verifying if the twenty percent discount is genuinely applied in the Product class if the product type is a book, according to the stated requirements listed previously in this section. The test creates an instance of the Product class, specifies a price of twenty as a decimal, runs the method to apply the discount, and stores the result in a variable. Finally, line 23

compares the result with the expected result, which is twenty, considering the given input. To run the tests on Visual Studio, you must choose the options Test on the superior menu and click on the “Run All Tests” option, as shown in *Figure 10.6*:

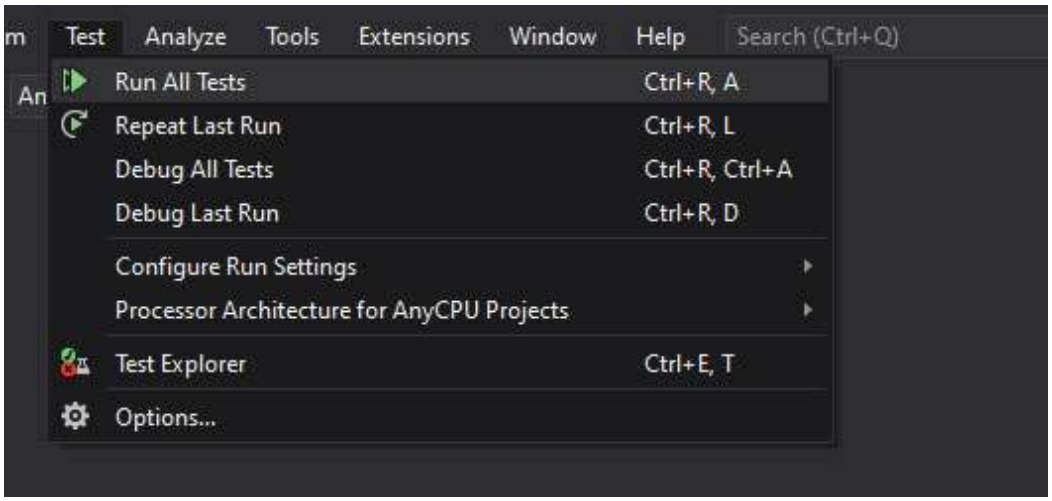


Figure 10.6: *Test menu option*

After that, the Visual Studio will open a specific window showing all the tests found in the solution, separated into classes and methods. The test running option executes the tests sequentially by default and considering all the comparisons between expected and actual results, in case of the conditions are satisfied, the Visual Studio shows the tests as green, indicating that the tests passed, as shown in *Figure 10.7*:

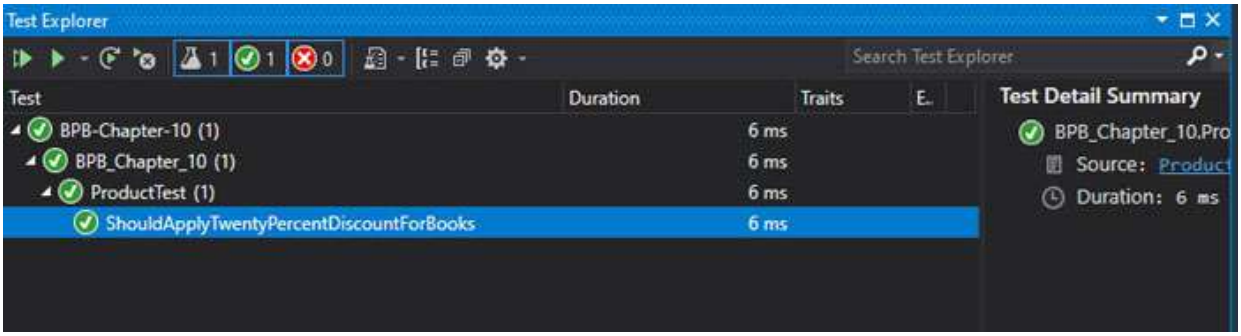


Figure 10.7: *Test window*

The test windows show detailed information on the time execution for each individual test, and considering the test was implemented correctly, it is marked as passed. It is really important in any unit test implementation to have a good code review, possibly by another developer, to ensure that it is

implemented correctly. A green test does not necessarily mean that the test is correct or meaningful, following all the desired requirements. If the test is correct, but the expected and actual results show different results, the test window highlights the test with the red color, as shown in *Figure 10.8*:

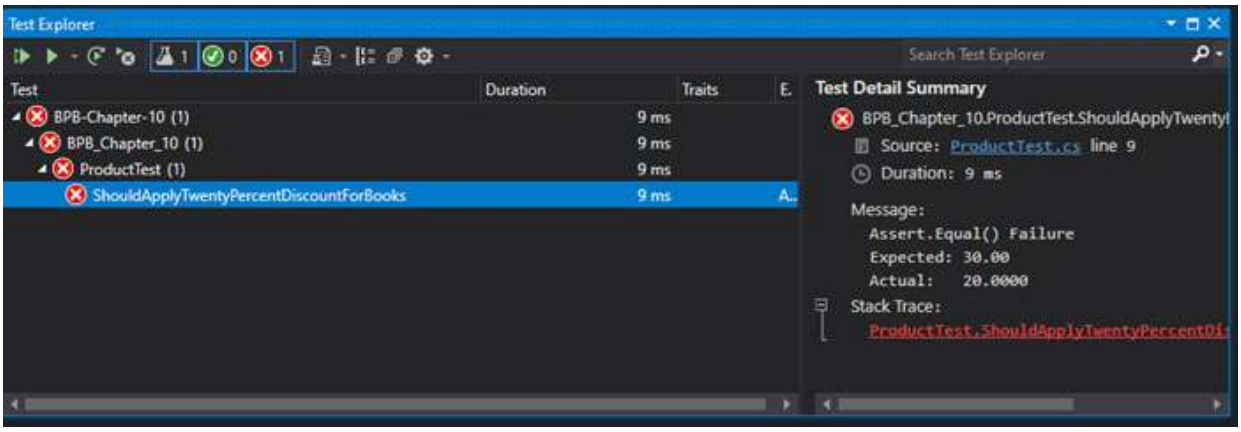


Figure 10.8: *A failed test example*

If the test fails, the window shows the error, indicating the expected and actual value found by the test itself. This situation requires reviewing the test and the class or method being tested. To force an error and show the results, you can change the expected result to a value different from twenty, which is clearly not what is specified in the Product class regarding discount for the book product type.

Unit tests, in general, should have good coverage for each class in the project to ensure all the possibilities and paths are correctly tested and to guarantee that changes in the code over time will not cause unexpected errors in existing functionalities. In the product class, the apply discount method has conditions for the customer’s anniversary and a path to return the original value if the product type is not a book and the customer's birthday does not correspond to the actual day. Therefore, extra test methods must be created to verify if the application discount method has the correct implementation. Following the requirements, the following condition to be tested is regarding the customer anniversary, which requires the creation of a new instance of a customer with the correct data, as shown in *Figure 10.9*:

```
27      [Fact]
        0 references
28      public void ShouldApplyThirtyPercentDiscountCustomerAnniversary()
29      {
30          Product product = new Product()
31          {
32              Id = 1,
33              IsOnlineProduct = false,
34              ProductType = ProductType.Book,
35              Price = 100.00m,
36              Title = "Test product"
37          };
38
39          Customer customer = new Customer();
40          customer.Birthday = DateTime.Now.AddYears(-30);
41
42          var result = product.ApplyDiscount(customer);
43          Assert.Equal(70.00m, result);
44
45      }
```

Figure 10.9: *Customer anniversary test*

In that case, the test method states that the customer was born thirty years ago from the actual date. As the price of the product is one hundred, and the expected result is a thirty percent discount, the expected product price is seventy. If the run test option is used again, the test window will show the complete list of the test, which indicates if they pass or not, as shown in *Figure 10.10*:

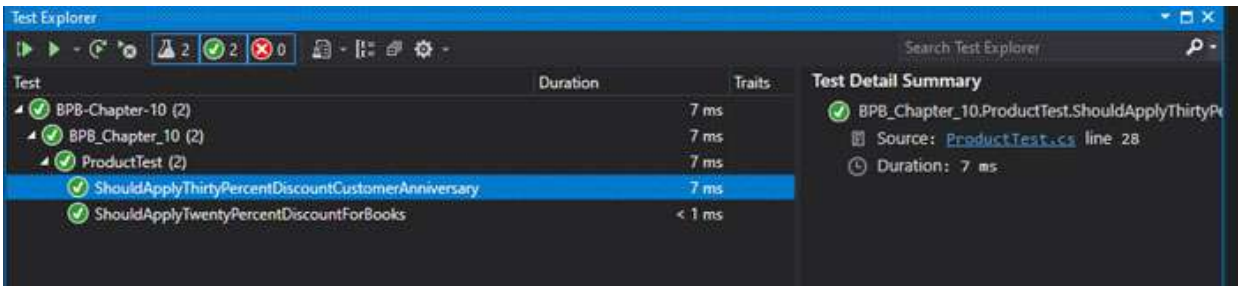


Figure 10.10: *Test result for customer anniversary test*

The test methods must have meaningful names following strictly what the test tries to verify the target class. To have relevant coverage, in many cases in real scenarios, the number of test methods is much greater than the methods in a class to test all the possibilities thoroughly. Therefore, the relation between class methods and test methods is not necessarily one-to-

one. The complexity of the target classes determines the number of necessary tests, and the recommended code coverage by tests usually are determined and stated project by project, following business prioritization, developer-hour costs, and the judgment of how certain critical functionality is for the project's success.

The xUnit tool allows us to specify annotations to input data in the test method using the attribute "**`InlineData`**" combined with the "**`Theory`**" attribute. In the previous examples, an instance of a product was created, passing a single price value, which means that the methods were tested only with one variation. To test the same method with multiple distinct values, it is possible to specify multiple values that must correspond to the method's parameters. This approach represents an excellent alternative to inject data into the method and reduce the effort in writing unit tests:

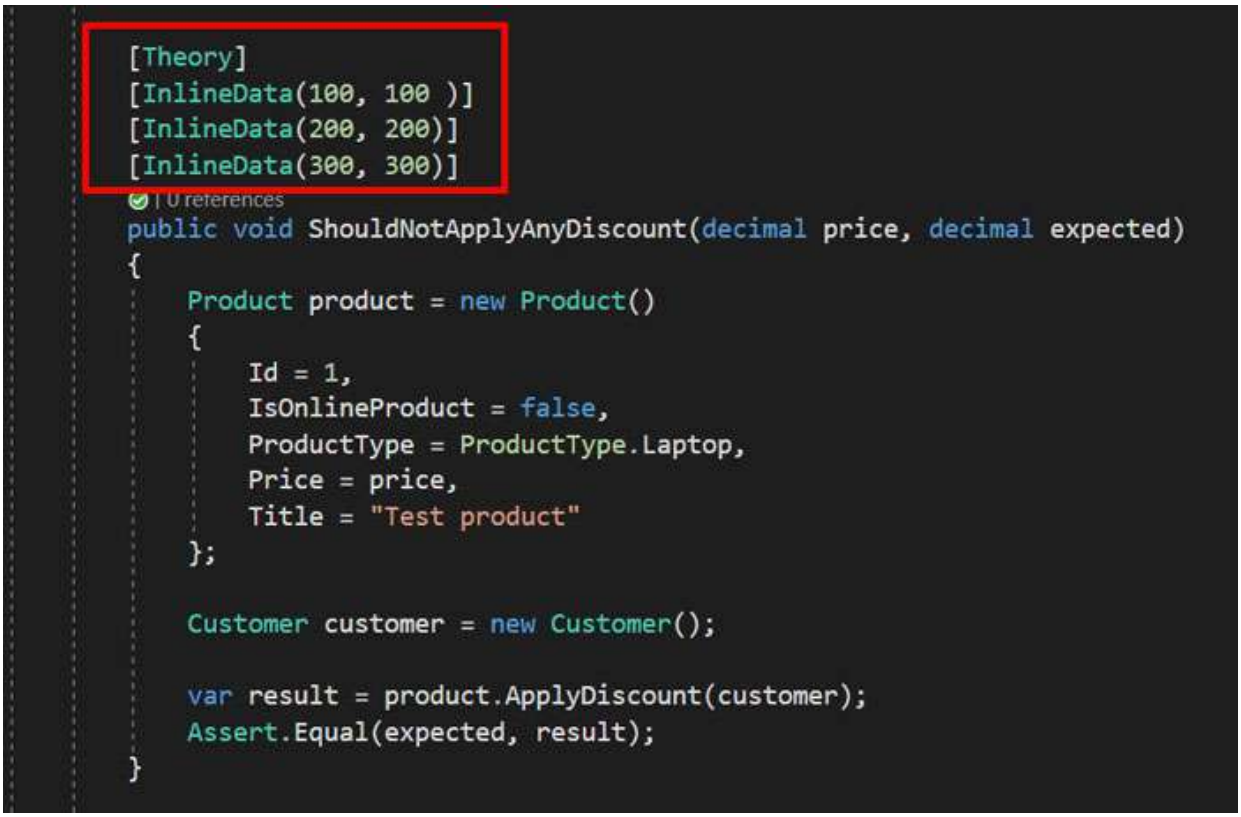


Figure 10.11: *Inline data attribute*

As multiple values are being passed to the method in the annotation attribute, the test will run three times by comparing the expected and actual results. It will present an error if at least one variation fails. After running the

tests, the test window shows the results, including the number of times that the method was executed, as shown in *Figure 10.12*:

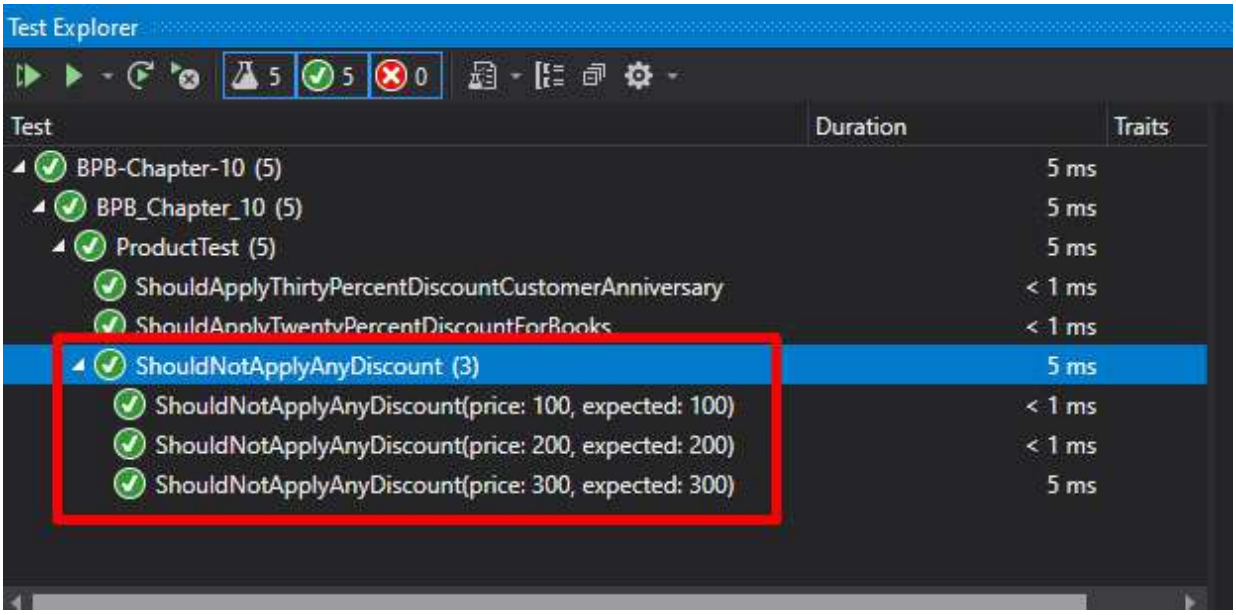


Figure 10.12: *Inline data attribute*

The xUnit provides many extension methods to verify the expected results in many ways, including verifying if a specific exception is thrown and checking whether a method or routine returns the correct type using reflection. This tool is broadly used in .NET projects and represents a reference for flexibility and extensibility as an open-source project.

Test-driven development (TDD)

In the middle of a development process, everyone involved in a software project is usually focused on meeting strict deadlines, monitoring costs, and implementing the stated scope. It is pretty common to leave the implementation of tests for the final of development, considering the system requirements are constantly changing. The tests are often made only during the short window available at the project's final. Considering the huge importance of automated tests to ensure quality and reliability, it is imperative to prioritize the implementation of tests as part of an essential part of the development process by any developer involved in a project. Many professionals in the market think that the inclusion of mandatory automated tests in planning is not a reachable goal once many projects have issues in implementation. However, implementing tests at the beginning of

the project is an excellent opportunity to make things right in the first moment and avoid further and future problems in production environments. The costs of software correction are much higher when applied lately in a project, and the necessary effort to implement automated tests in legacy projects is much more complex.

In fact, implementing tests before the actual code implementation helps us write better code. During the development, tests are much easier to identify architectural and implementation problems considering a class or method must be testable and follow the good practices of software development, including the observance of recommended practices of object-oriented programming paradigm, such as SOLID principles, which were already overviewed previously in this book. The test-driven development consists of always writing the test before any actual implementation. In the scenario exemplified in the previous section, it would mean writing the unit tests for the product class before the full implementation of the product class itself. Considering the requirements regarding discount, the apply discount method would have the same tests, but the implementation would not have any conditions, as shown in *Figure 10.13*:

```
7 references
public class Product
{
    3 references | 5/5 passing
    public int Id { get; set; }
    3 references | 5/5 passing
    public string Title { get; set; }
    4 references | 5/5 passing
    public decimal Price { get; set; }
    3 references | 5/5 passing
    public ProductType ProductType { get; set; }
    4 references | 5/5 passing
    public bool IsOnlineProduct { get; set; }

    3 references | 5/5 passing
    public decimal ApplyDiscount(Customer customer)
    {

        return this.Price;
    }
}
```

Figure 10.13: *Apply discount method without conditions*

In that case, if the test methods were implemented before, the tests would intentionally fail until the necessary code is implemented in the product class, as shown in *Figure 10.14*:

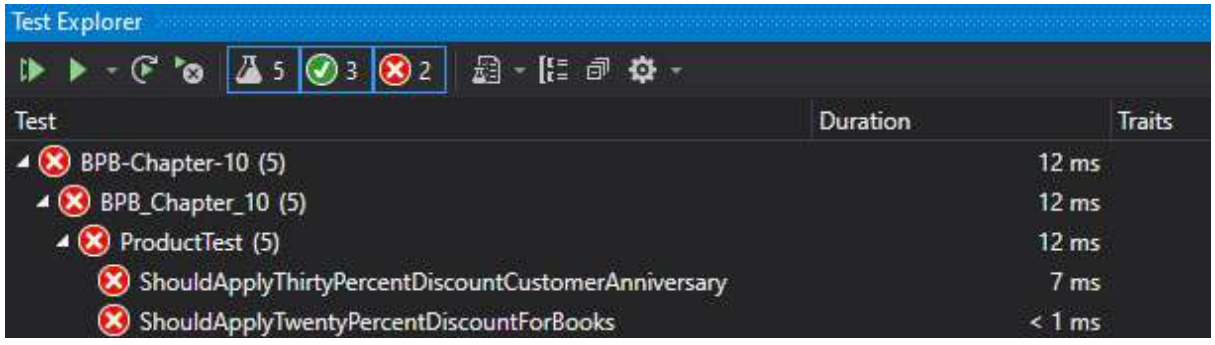


Figure 10.14: *Faled tests*

Writing the tests before the target class's implementation allows us to review the software requirements and ensure that valid tests are safely verifying the

implementation. In the preceding image, considering the apply discount method does not have the necessary code to apply the custom discount for the book product type and based on the customer anniversary, the test fails until the rules are correctly implemented. Basically, the **test-driven development** (**TDD**) approach consists of the following steps:

1. Write the tests.
2. Write the necessary code.
3. Refactor until the test pass.

This approach requires a mind-changing for development teams that usually do not take this way and also requires maturity in terms of organizational culture and agile methods, being entirely beyond only technical aspects of a software development technique.

Points to Remember

- The XUnit is a powerful open-source testing tool for .NET projects.
- The classes and methods must be implemented to be testable and easy to understand, following the good practices of object-oriented programming.
- It is highly recommended to write the unit tests before the code itself to have good coverage of tests in the project.

Conclusion

As seen in this chapter, the xUnit tool provides many features to properly test classes and methods in C# language and other languages supported by the .NET platform. The unit tests are usually based on the comparison between actual and expected results, and it is possible to execute multiple variations and run tests using the Visual Studio IDE, keeping a good quality of the code while the coding process is being made. The test-driven development technique allows us to write tests at the beginning of the development process, implement code safely, and ensure that new issues will not be introduced in future implementations.

In this chapter, you learned essential and fundamental concepts of unit tests in C# language using the xUnit tool. Additionally, you had the opportunity to

understand the **Test-Driven Development** (**TDD**) approach, create unit tests from scratch using a hypothetical scenario, and familiarize yourself with unit tests.

In the next chapter, you will have the chance to learn the new features introduced in C# 8.0 9.0 and a brief overview of C# 10.

Multiple Choice Questions

1. **Which annotation attribute can be used for the compiler to identify if a method is a test?**
    a. Fact
    b. Test
    c. Equal
    d. Assert
2. **Which annotation can be used to inject data into a test method?**
    a. Parameter
    b. Fact
    c. Theory and InlineData
    d. None of them
3. **Which alternative does not represent a valid assert method?**
    a. True()
    b. Equal()
    c. EndsWith()
    d. NotTrue()

Answers

1. **a**
2. **c**
3. **d**

Questions

1. Explain the main purpose of unit tests.
2. Using the knowledge, you have gained in this chapter, create unit tests for the Order class presented in this chapter.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11

New Features in C# 8.0 and 9.0

Introduction

In this chapter, we will walk you through the most relevant features available for C# language introduced in versions 8.0 and 9.0, including performance improvements presented since the .NET Core 3.0.

The chapter contains examples that allows you to learn switch expressions, new asynchronous disposable possibilities, static local functions, default interface methods, and other valuable changes in the latest C# language.

Getting familiar with the new features in the C# language is essential to update existing projects based on .NET Core for increasing the performance of the software, mainly in asynchronous operations, and to have a chance to apply more efficient statements in your daily code, improving your developer skills using the most modern version of C# language in real projects.

Structure

In this chapter, we will discuss the following topics:

- New features in C# 8.0
- New features in C# 9.0

Objectives

After studying this unit, we will be able to use the new features of C# 8.0 and C# 9.0. We will also learn to apply the refactoring process in legacy projects and identify performance improvements aspects that can be applied in legacy projects regarding the C# language

New features in C# 8.0

Until this point, you learned the fundamental existing features available in the C# language that was introduced to it. Since the very first versions at the beginning of the .NET Framework to familiarize you with the basic programming language concepts.

Additionally, in this book, you had the chance to learn and apply practical examples of essential object-oriented programming concepts, including SOLID principles and Design Patterns. Now, it is time to learn the most fundamental improvements introduced in C# language in the most recently released versions, which allows us to understand not only to keep our projects based on the .NET platform posted but also gives us a chance to get the advantage of all performance improvements in the most modern version of C# language.

All the changes shown in this current section are supported since the .NET Core 3 version, and it is also available since .NET Standard 2.1 is compatible with all the subsequent versions. Therefore, it is necessary to have the most recent production version of .NET Core to run all the examples of this section correctly. The chapter does not contain all the changes available in the C# language, but it is focused on the most used features, being related to the most common scenarios presented in daily projects.

One of the most common new features introduced in C# 8.0 is regarding the new possibilities for switch expressions that allow us to specify the switch statement combined with a variable assignment, as shown in *Figure 11.1*:

```
14  static void SwitchExpression()
15  {
16      string messageCustomer = DateTime.Now.DayOfWeek switch
17      {
18          DayOfWeek.Monday => "The promotion is not available",
19          DayOfWeek.Tuesday => "The promotion is available only for Laptops",
20          DayOfWeek.Wednesday => "The promotion is available for all products",
21          DayOfWeek.Thursday => "The promotion is available only for Books",
22          DayOfWeek.Friday => "You are lucky. It is Friday and all the products have twenty percent discount"
23      };
24
25
26      Console.WriteLine(messageCustomer);
27      Console.ReadLine();
28  }
```

Figure 11.1: *Switch expressions example*

This new way to specify switch expressions represents a more readable code in cases where the result of the statement must be a variable assignment. Furthermore, as seen in the preceding image, the use of "**`break`**" is not

necessary as it is applicable only for traditional switch statements. Another possibility of switch expressions in the new version of C# language is the use of the keyword "_" to replace the legacy default option, as shown in *Figure 11.2*:

```
string messageCustomer = DateTime.Now.DayOfWeek switch
{
    DayOfWeek.Monday => "The promotion is not available",
    DayOfWeek.Tuesday => "The promotion is available only for Laptops",
    DayOfWeek.Wednesday => "The promotion is available for all products",
    DayOfWeek.Thursday => "The promotion is available only for Books",
    DayOfWeek.Friday => "You are lucky. It is Friday and all the products have twenty percent discount",
    _ => "None of the products has discount now"
};
```

Figure 11.2: *Default option for switch expressions*

The following examples are regarding the possibility of specifying default implementation for interface methods, which needs to be used carefully in real projects following the best practices of interface segregation principle and respecting the primary purpose of interfaces in real projects. This new feature represents an alternative to the abstract class concept in many cases, mainly in scenarios where only a few child classes have a custom implementation for specific methods. The use of **default interface methods** can be implemented, as shown in *Figure 11.3*:

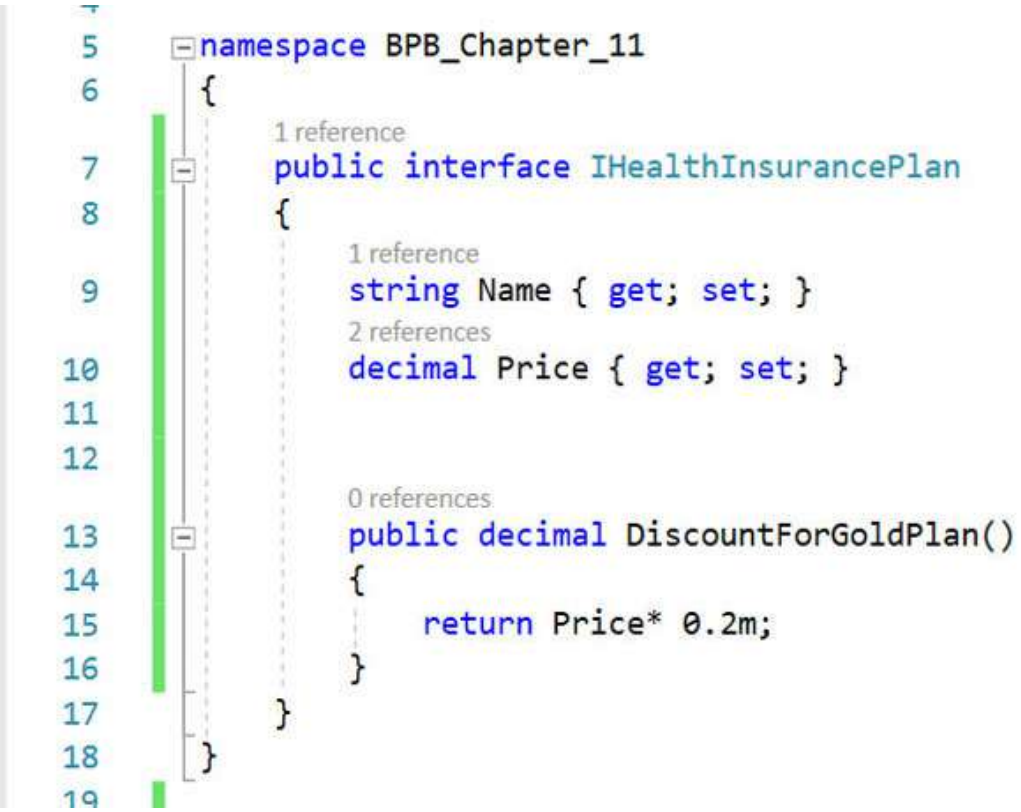


Figure 11.3: *Default option for switch expressions*

With this new feature, a class that implements the interface does not need to explicitly specify the method that already has an implementation in the interface, contrasting to the previous version where the compile did not allow this operation, being mandatory before creating the method in the implementation class. In the case where the implementation is practically the same for the majority of classes, the use of this feature is beneficial. It allows us to keep clean classes and avoid using abstract classes when an interface is more suitable. In this new version of interfaces, the compile does not show any error if the method is not specified in the implementation class for default methods, as shown in *Figure 11.4*:

```
5   namespace BPB_Chapter_11
6   {
        0 references
7       public class MedicalPlanForPrivateCompanies : IHealthInsurancePlan
8       {
            1 reference
9           public string Name { get; set; }
10
            2 references
11          public decimal Price { get; set; }
12      }
13  }
```

Figure 11.4: *Implementation class example*

The following relevant improvement introduced in C# 8.0 is regarding the "**`using`**" statement. In the previous version, it was necessary to tell the compile that a specific variable needs to be disposed of after the operations inside a particular statement are complete. This approach is highly recommended for external resource manipulation, such as database connection and files. However, the previous version of the "**`using`**" statement is hard to read in many cases in a code maintenance process if the routine requires many interpolate statements. Since C# 8.0, it is possible to specify the "**`using`**" keyword before a variable creation, as shown in *Figure 11.5*:

```
        1 reference
65      static void NewUsingImplementation()
66      {
67          using StreamReader quotationFile =
68              new StreamReader("quotation.txt");
69
70          string quotation;
71
72          while ((quotation = quotationFile.ReadLine()) != null)
73          {
74              Console.WriteLine("Quotation: " + quotation);
75          }
76
77      }
```

Figure 11.5: *"Using" declaration*

In that case, the object created with the "**`using`**" keyword will be automatically disposed of after the current scope, which means when the

method execution is complete. Compared with the old way of disposing of objects, this form represents a better approach if the object does not need to be released from the memory before the end of the method execution. The following image represents the same code but uses the old implementation of the "**`using`**" statement:

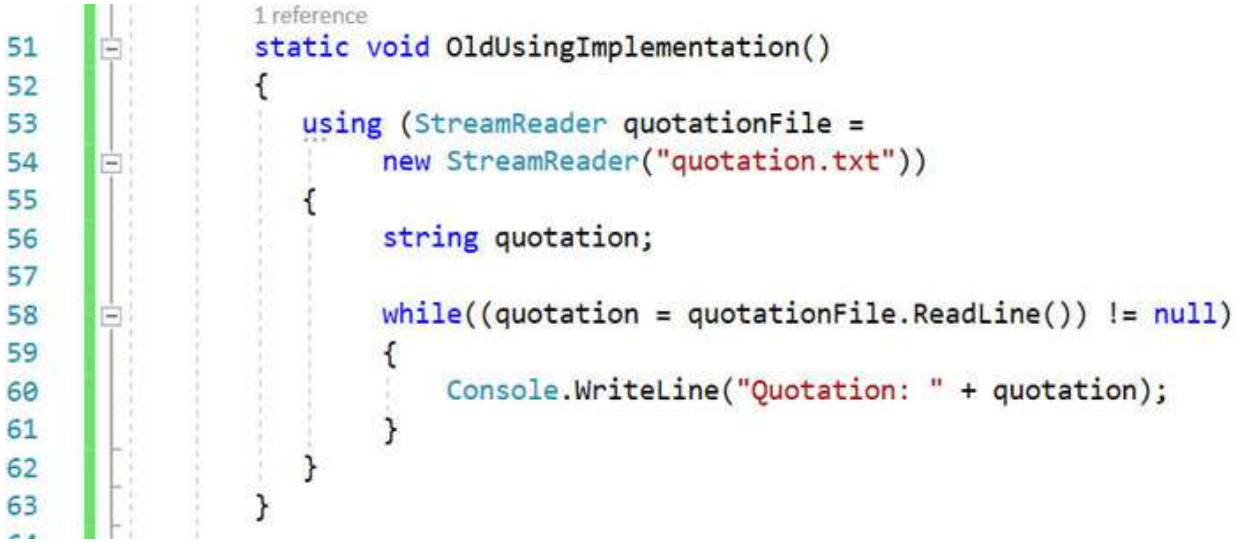


Figure 11.6: *Old using statement*

The old option is still valid in that case, and the applicability of the new implementation depends on the code decision process to follow the approach that makes more sense in your specific scenario.

The most recent new versions of C# brought to our application's significant improvements in terms of performance and asynchronous operations, thus allowing us to take advantage of using the full potential of processors with multiple cores in production environments, giving to users a better experience in terms of request loading, mainly for Web applications. C# 8.0 introduced the possibility of using an asynchronous concept combined with enumerable objects, increasing the performance of many common routines requiring iterating an object. To use that feature, it is necessary to implement the interface "IAsyncEnumerable<T>" as shown in the following code:

```
1 reference
private static async IAsyncEnumerable<decimal> GetPricesAsync()
{
    foreach (var healthPlan in GetHealthPlans())
    {
        decimal price = await new HealthPlanService().GetPrice(healthPlan);
        yield return price;
    }
}
```

Figure 11.7: *IAsyncEnumerable<T> interface*

In this example, the method "**`GetPriceAsync`** " returns a list of prices after calling multiple times an async method in the Health Plan Service class. As this is an asynchronous operation, once the primary method is called, the application does not need to wait for all the prices, being possible to process those simultaneously using multi-threads. This represents a considerable improvement to the previous versions of C# language that supported only synchronous operations for loops in general. To consume an async endurable, it is necessary to use the operation "**`await`**" combined with the "**`foreach`**" declaration, as shown in *Figure 11.8*:

```
0 references
private static async Task AsyncStreamExample()
{
    await foreach(var price in GetPricesAsync())
    {
        Console.WriteLine("Price: " + price);
    }
}
```

Figure 11.8: *Async "foreach"*

In this example, the price is printed to the console asynchronously once each item is released by the GetPriceAsync method. The Async Stream implementation in C# language states one of the best improvements in .NET applications, being possible to process streams taking the full power of a server, running many multiple threads at the same according to the limit of the server configuration.

Array and index manipulations are pretty common in any software, from the basic to the most complex ones. C# 8.0 brought new features to work with ranges and indexes, giving simplicity and an extra alternative to having a

more expressive and clear code. To access a specific index, mainly in the final of the list, it is not necessary to use the keyword length, being possible to use the "^" keyword to access the inverted index, as shown in the following code:

```
1 reference
private static void IndicesAndRangeExample()
{
    var healthPlans = new string[]
    {
        "Plan1",
        "Plan2",
        "Plan3",
        "Plan4",
        "Plan5",
        "Plan6",
        "Plan7",
        "Plan8",
        "Plan9",
        "Plan10"
    };

    var firstPlan = healthPlans[0];
    var lastPlan = healthPlans[^1];

    Console.WriteLine("First plan: " + firstPlan);
    Console.WriteLine("Last plan: " + lastPlan);
}
```

Figure 11.9: *Index*

The method assigns the first and last occurrence in the health plan array two different variables. With the use "^" operator, it is not mandatory to check the length of the array to avoid exceptions and unexpected behavior of array

manipulation. If we run the method in a Console application, the result will like the following image:

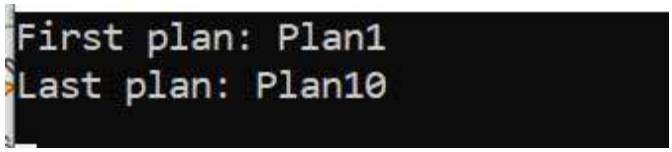


Figure 11.10: *Index result*

Therefore, the new feature produces the expected result, following the recommendations in terms of simplicity recommended by the technical community and companies regarding desired improvements for C# language. Another extra possibility is the new possibilities in terms of accessing a specific range of an array, as exemplified in *Figure 11.11*:

```
var healthPlans = new string[]
{
    "Plan1",
    "Plan2",
    "Plan3",
    "Plan4",
    "Plan5",
    "Plan6",
    "Plan7",
    "Plan8",
    "Plan9",
    "Plan10"
};

//Plan1, Plan2, Plan3, Plan4
var firstFourPLans = healthPlans[..4];

var allPlans = healthPlans[..];

//Plan8, Plan9, Plan10
var lastThreePlans = healthPlans[^3..];
```

Figure 11.11: *Range example*

Among all the improvements presented in C# 8.0, the possibility of asynchronous for enumerable is the most expressive one because since the first version of .NET Core, the performance has been a key point for the entire platform, and it is possible to clearly see the vast improvements in terms of memory usage and processor consumption. Since you learned essential new features introduced in C# 8.0, in the next section, you will have the opportunity to familiarize yourself with all the new features available in C# 9.0.

New features in C# 9.0

C# 9.0 introduced to the .NET platform relevant improvements in object initialization, simplicity, and new ways to handle data across the system, giving more possibilities regarding encapsulation and data protection to be used in more advanced scenarios, despite the syntax being quite simple. Data protection in classes usually requires extra code, the use of additional constructors, and a lot of concerns in terms of testing. With C# 9.0, it is possible to protect class properties from modification without changing the class structure. Creating different methods to achieve similar results as the previous language version is no longer necessary.

First of all, one of the most valuable features in the new version is regarding the Init Accessor, which allows us to specify the mutability of a class member explicitly, which indicates to the compile that a specific property can be changed only when the class is created. Additionally, it is not necessary anymore to specify the constructor in that case as a method, being possible to initialize the class differently and use the keyword “init” instead of “set” to become the property immutable after initialization, as shown in *Figure 11.12*:

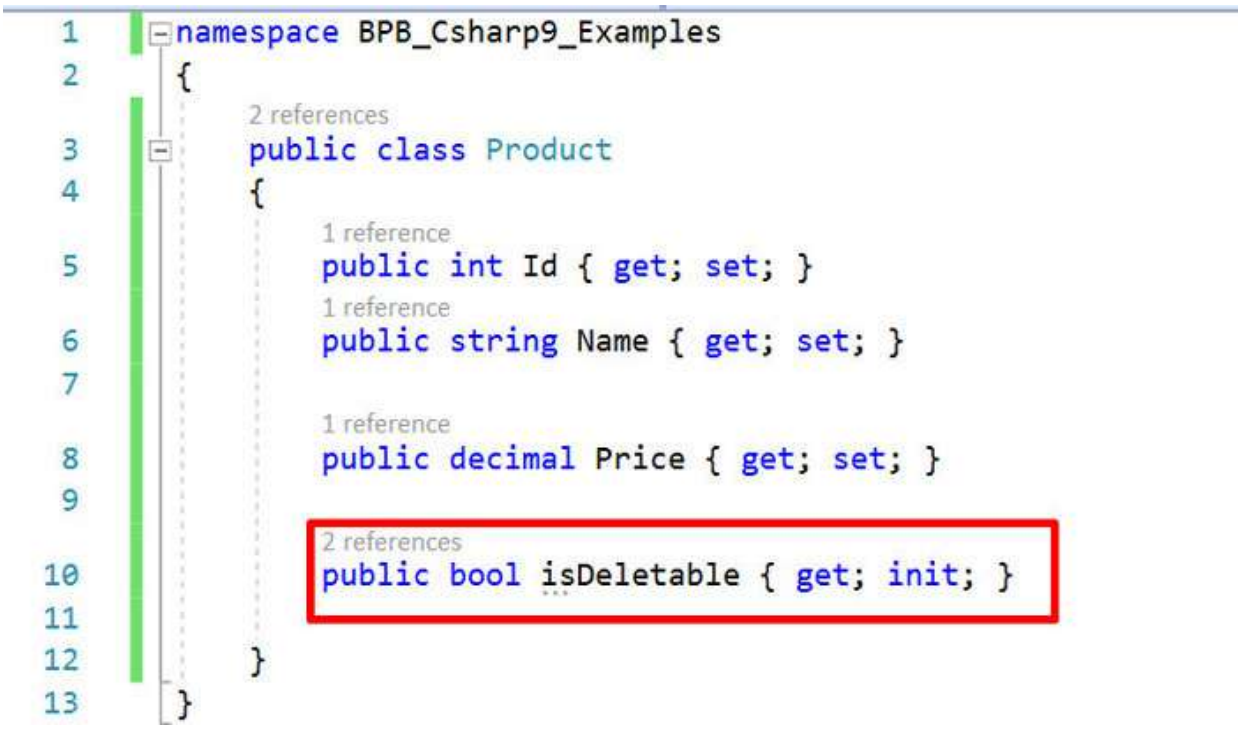


Figure 11.12: *Init accessor*

In this example, as the "**`isDeletable`**" property uses the init accessor, the assignment to its value is allowed only in the object initialization, and the compilation will show an error in case of after the value changes after the object creation, as shown in *Figure 11.13*:

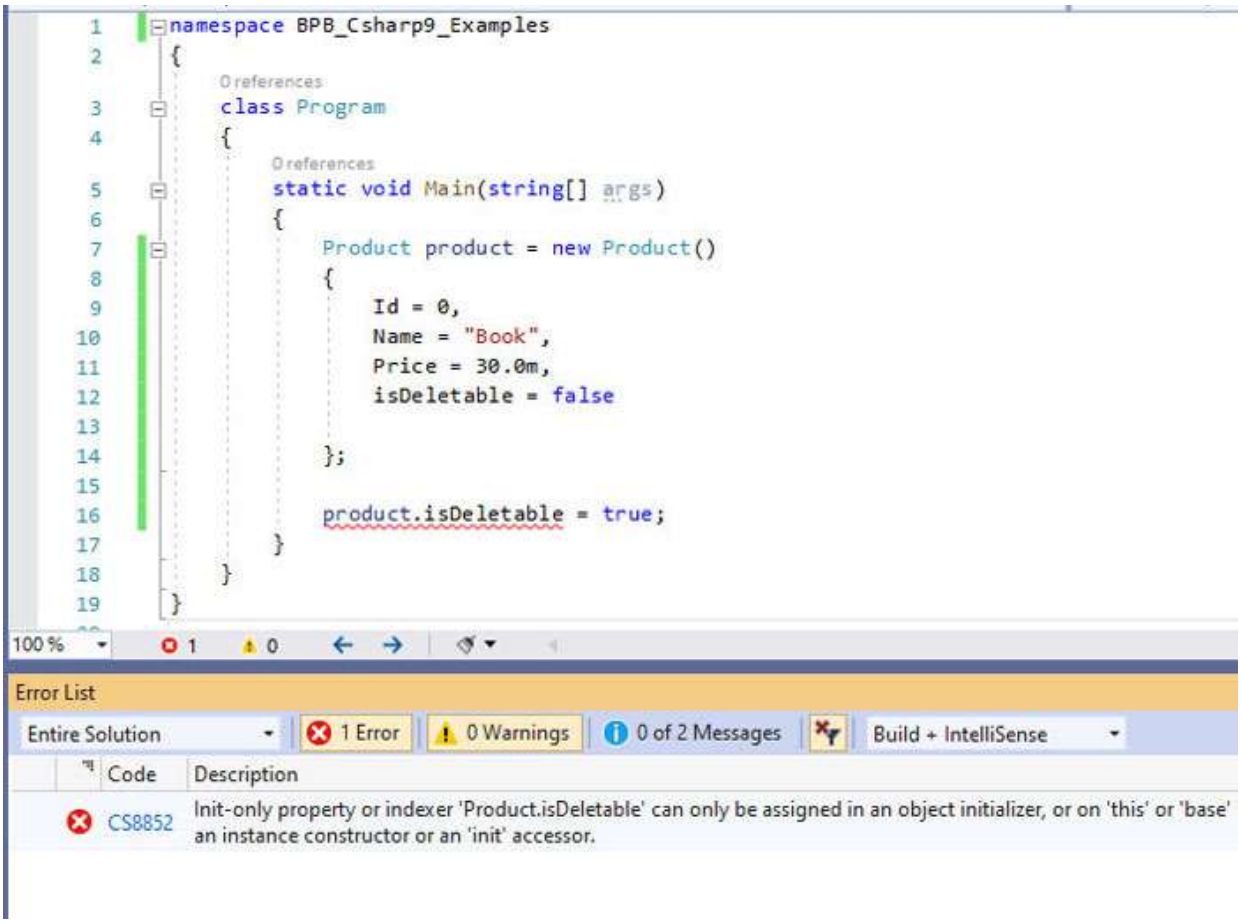


Figure 11.13: *Init accessor*

This approach is advantageous because it achieves the purpose of protecting the property from undesired modification at the same time as the class structure is kept as simple as it should be. Therefore, to create immutable properties, it is no longer mandatory to specify extra constructors passing in the parameters of object property values combined with private properties.

In certain scenarios, a specific implementation requires to become all the properties in a class to be immutable for different reasons, such as bringing records from a database or data that comes from an external system using APIs, among other motivations in keeping the object protected from modification. If that is the case, C# 9.0 contains a new class modifier called “record,” which automatically changes to immutable all the properties and marks the class as a record, protecting the whole class from undesired modification. The data modifier can be used as shown in *Figure 11.14*:

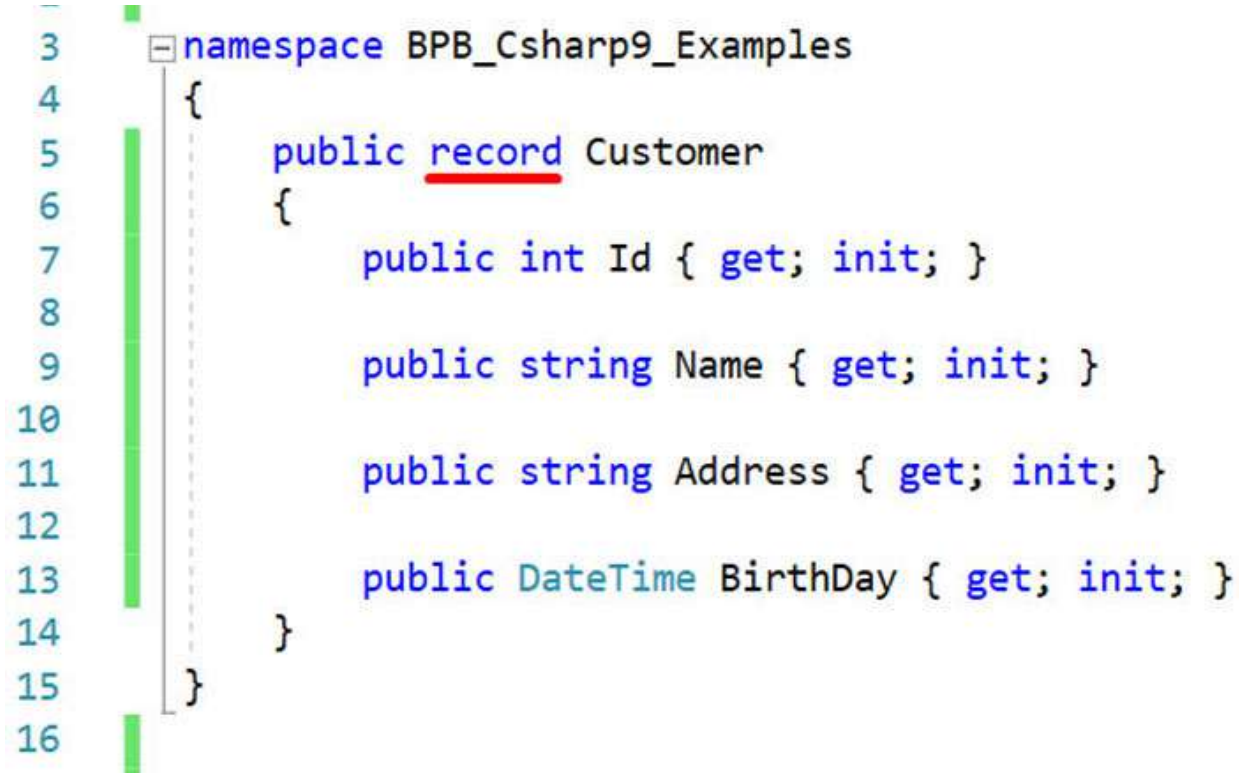


Figure 11.14: *Record modifier*

Using the "**`record`**" gives us extra possibilities in terms of value and data manipulation, and it also represents a gain in performance. Combined with records, it can use other keywords to make a copy and correctly compare objects. The "**`with`**" keyword allows us to create a copy of an object, changing only a few properties if it is necessary, as shown in *Figure 11.15*:

```
0 references
static void Main(string[] args)
{
    Customer customer1 = new Customer()
    {
        Id = 1,
        Name = "Customer 1",
        BirthDay = new DateTime(day: 01, month: 12, year: 1987),
        Address = "Address of the customer 1"
    };

    var customer2 = customer1 with { Name = "Customer 2" };

    Console.WriteLine(customer2);
}
}
```

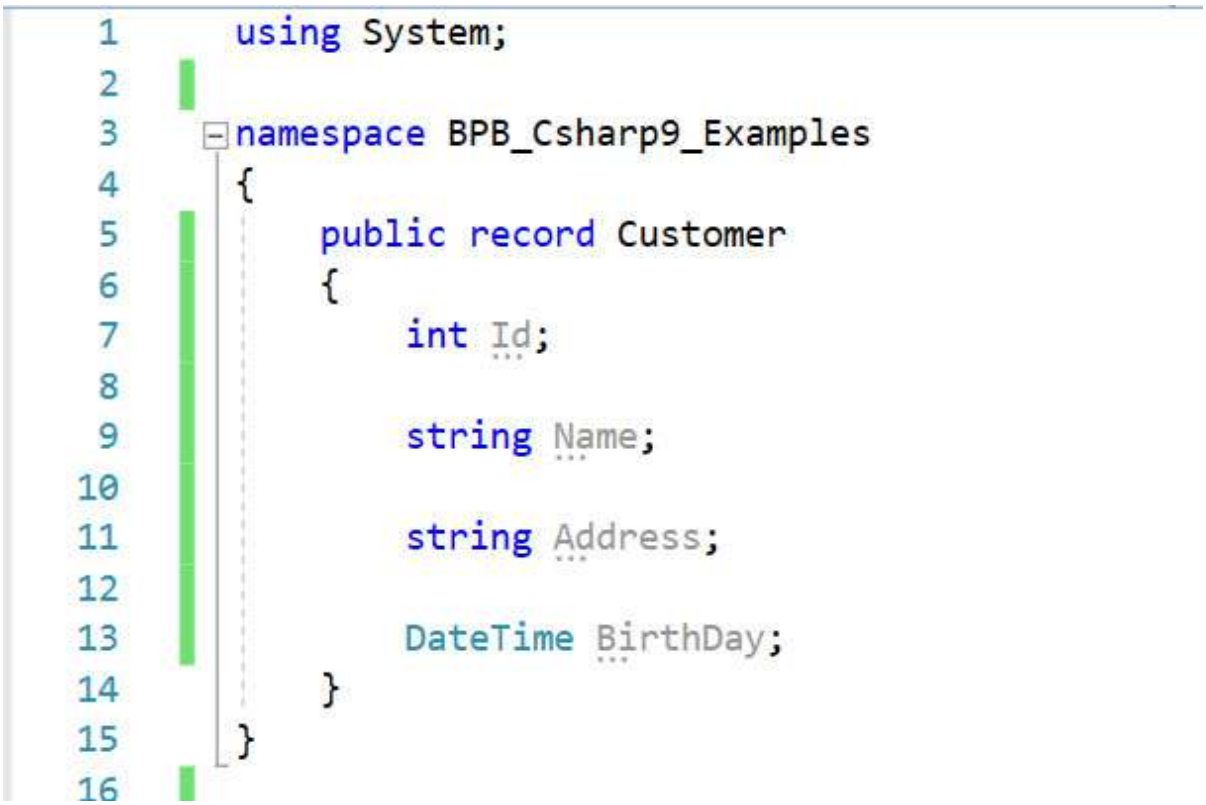
Figure 11.15: *Use of the "with" keyword*

Additionally, using records allows us to work appropriately with comparisons between objects, which can represent a considerable challenge in specific complex scenarios using the previous versions of the C# language. If comparison needs to be made, the “Equals” method works as expected without any extra implementation, as shown in *Figure 11.16*:

```
Customer customer1 = new Customer()
{
    Id = 1,
    Name = "Customer 1",
    BirthDay = new DateTime(day: 01, month: 12, year: 1987),
    Address = "Address of the customer 1"
};

Customer customer2 = new Customer()
{
    Id = 1,
    Name = "Customer 1",
    BirthDay = new DateTime(day: 01, month: 12, year: 1987),
    Address = "Address of the customer 1"
};

Console.WriteLine("Returns true: " + customer1.Equals(customer2));
Console.WriteLine("Returns false: " + (customer1 == customer2));
```

Figure 11.16: *Comparison between objects*

In this example, the “customer1” and “customer2’ objects have the same values in all the properties. However, only the " Equals " comparison returns equality as true. Finally, another quite helpful feature introduced in C# 9.0 regarding records is the possibility of declaring properties merely, as shown in the following code:



```
1   using System;
2
3   namespace BPB_Csharp9_Examples
4   {
5       public record Customer
6       {
7           int Id;
8
9           string Name;
10
11          string Address;
12
13          DateTime BirthDay;
14      }
15  }
16
```

Figure 11.17: *Simpler properties declaration*

In that case, the properties without modifiers become a private members by default, not being possible to change the value after. The records in C# language allow us to work with inheritance as well, which might be helpful in scenarios where complex data needs to be handled by external resources.

C# 9.0 contains other changes regarding new features and performance improvements, but this section is limited only to the changes regarding classes. Considering the C# 9.0 final release is still in progress until the moment when this book was written, it is recommended to check the official C# documentation provided by Microsoft for updates.

Points to remember

- C# 8.0 introduced the possibility of having interfaces with a default implementation for methods.
- The record keyword allows us to create read-only objects, increasing the system's performance.
- The init accessor can be used to become property as read-only.

- With the new features in C# 8.0, it can work with asynchronous streams.

Conclusion

As seen in this chapter, the C# language received many significant improvements and new features, simplifying the way to build classes and increasing the quality of testability and readability in complex scenarios. The 8.0 and 9.0 C# versions represent the central point of all the improvements presented since the first version of .NET Core in terms of simplicity and good performance. All the changes were made based on the technical community feedback; some already existed in other modern languages. Therefore, the C# language is becoming much more powerful and compliant with market needs.

In this chapter, you learned the most recent changes in C# language, including improvements on switch expressions, asynchronous streams, and new ways to handle indexes and ranges. Additionally, you had the opportunity to familiarize yourself with the init accessor in C# 8.0. You learned how to work with records to make the classes simpler and well-controlled in terms of protection.

In the next chapter, you will have the chance to learn how to build applications for LINUX, including universal executable creation for desktop applications.

Multiple Choice Questions

1. **Which keyword can be used in C# to become a whole class as read-only?**

    a. read-only

    b. record

    c. immutable

    d. struct

2. **What keyword can be used to become a class property as read-only:**

    a. public

b. protected

c. private

d. init

3. **Which alternative is valid for object comparison in C# 8.0 for records?**

a. ==

b. Equals()

c. !=

d. IsTrue()

Answers

1. **b**
2. **d**
3. **b**

Questions

1. Build the implementation of the switch expression shown previously in this chapter.
2. Using the knowledge you gained in this chapter, build a series of record structures.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 12

Building .NET Applications for Linux

Introduction

In this chapter, we will walk you through the most relevant features in .NET for cross-platform development using Linux, including examples in terms of possibilities and restrictions.

The chapter is full of considerations on development for Linux distributions, giving you the opportunity to know the scenarios where the development of .NET applications for other operating systems is suitable for large-scale applications.

Getting familiar with the development of .NET applications for Linux is essential to understand the cross-platform aspect of the most recent versions of .NET, mainly for saving operating systems and server infrastructure costs, including crucial aspects of system security for large-scale projects.

Structure

In this chapter, we will discuss the following topics:

- Advantages of multi-platform systems
- .NET projects available for Linux
- Developing .NET applications with WSL 2

Objectives

After studying this unit, you should be able to understand multi-platform concepts, create your first .NET application for Linux and comprehend the pros and cons of .NET applications for Linux. You will also be able to use WSL to develop .NET applications

Advantages of multi-platform concepts

Considering the diverse environment presented in the market in terms of technologies, programming languages, platforms, and devices, most prominent tech companies understand the essential and urgent need for the modernization of their business to support cross-platform development.

Cross-platform development states that a single software must run adequately in multiple operating systems, such as Windows, Linux, and macOS, including distributions for mobile devices. The achievement of this objective represents a considerable challenge for any tech company and developers of business applications in general, as not all the programming languages and platforms are compatible with multiple operating systems.

Apple allows the development of applications for macOS using Swift and Objective-C, programming languages that are not natively compatible with Windows and Linux operating systems; despite the fact, it is possible to run on those operating systems using GNUstep and Objective-C compilers for development purposes. On the other hand, the first versions of the .NET platform were not compatible with Linux and macOS, limiting the popularization of the framework among Java developers and other cross-platform-friendly technologies.

Since the release of .NET Core 1.0, Microsoft has turned the way of developing applications using the .NET platform with the primary focus of unifying .NET APIs in general, but evidently with the extra intention of allowing cross-platform development using C#, Asp.Net, WebAssembly, including the possibility of installing Microsoft products such as SQL Server and Visual Studio in other different operating systems. The traditional Visual Studio is available for macOS, and Visual Studio Code is available for macOS and various LINUX distributions.

Furthermore, it is even possible to deploy .NET applications in Linux using .NET Core 1.0, 2.0, 3.0, and .NET 5, including Console Applications and Asp.Net Core applications. This characteristic of .NET helped increase the market for C# applications as many companies have the policy of only using Linux distributions for compliance and cost reasons.

The fact that .NET and Visual Studio are now available on a broader spectrum increases the potential of developers from other platforms to migrate to .NET development as they are able now to keep using almost any operating system without the need for configuring and maintaining costly

virtual machines just to run Windows operating systems to develop .NET applications.

In the following sections, you will explore the main benefits of cross-platform development for business applications, starting with market opportunities.

Market opportunities

Developing a large-scale system that would be available for a vast number of customers worldwide is a challenging and complex process involving high costs in terms of professional hiring, design, and architecture of enterprise applications, including costs regarding application deployment and maintainability. All these natural issues on software development have a much higher risk if the software is based on a single platform technology, which would require developing and deploying the applications in a single type of operating system, limiting the profitability that a diverse and competitive market provides. For example, if the software can be used in Linux, Windows, and macOS, it clearly represents a more competitive product in terms of market penetration as there are fewer restrictions.

The same applies to mobile applications, as using a technology that allows us to deploy and publish the application on multiple devices and platforms represents a considerable advantage in terms of development costs. Xamarin is an excellent example of reusability and cross-platform development. It is possible to use C# to develop applications published on Apple Store and Play Store, despite Apple and Android devices not being natively compatible with C#.

Maintainability

The possibility of using the same IDE for development in multiple operating systems using .NET and the flexibility of deploying the same application based on various platforms represent a turning point regarding hours spent in software development. Each platform may have its requirements, but .NET allows us to take advantage of cross-platform development using only C# language, allowing the fast migration of legacy projects built in old versions of .NET Framework to the most modern version of .NET, versions 5 and 6.

Hiring process

Many companies base their projects only on open-source technologies compatible with Linux and other operating systems, such as Python and Java. However, the restrictions of languages that can be used in a project also limit the possibility of hiring developers as the profile of professionals may vary in the market, from time to time, in terms of programming language and interests. Having cross-platform software to maintain makes the software company more competitive, increasing the chance of getting new developers.

Security

The use of cross-platform technology in software development brings us other essential benefits in terms of security. It allows us to have a more diverse infrastructure environment for deployment, making possible security vulnerabilities harder to find by intruders. If your software can be deployed in only one operating system, it reduces the complexity of hacker attacks. Furthermore, you can take advantage of all the evolution and modernization provided by the biggest players in the market in terms of compliance and security enhancements. This means that you have more options to choose from when you are designing an application.

System integrations

Developing a multi-platform system increases the integration possibilities with other applications and tools belonging to a particular operating system or platform. Despite the main reason for building a cross-platform being the broader range of options in terms of deployments, if the software can adapt to each operating system, it is possible to take the general benefits of the cross-platform application at the same time as providing specific features that would be possible and would be suitable only for a particular platform.

Fewer costs

Designing an application that works perfectly fine in multiple operating systems and runs smoothly on numerous devices is a massive challenge for any software company. The number of technologies available has increased exponentially over the last few years. The use of software for final users is not restricted anymore to conventional personal computers and cell phones. Nowadays, an unlimited number of devices require software and an

operating system to function properly, such as smartwatches, electric cars, airplanes, domestic electronic devices, and much more. The improvements in Internet connections worldwide and users' perception change regarding information availability force tech companies to adapt their development cycle to include support of any software to a more extensive range of devices and operating systems in the same proportion.

Considering this fact, the use of a cross-platform technology, such as .NET since the .NET Core 1.0 version and, more recently, .NET 6, is definitely a pivotal point in terms of cost management for the entire software development cycle as it allows you to create a single application that can run on multiple devices and operating systems. That means developing the software once instead of having multiple teams in the company; each of them maintaining and developing the same software for multiple platforms.

.NET projects available for Linux

Since the beginning of the .NET platform, desktop applications using Windows Form templates have become extremely popular in the market. It was combined with powerful visual design tools on Visual Studio for Windows, allowing developers to create complex applications in terms of the user interface quickly. It was possible to drag and drop components onto a screen and control the actions of buttons and other events in a significantly friendly way. Since the development of desktop applications based on Windows Forms was massively promoted over the last 15 years, many companies have kept their infrastructure environments focused on the Microsoft Windows operating system.

That situation contrasts with what has been happening in the market recently, as many companies adopt multi-platform systems. Windows Forms, as the name itself suggests, is supposed to run only in the Windows operating system. Despite all the effort put into the evolution and modernization of .NET by the releases of cross-platform versions since .NET Core 1.0, some project templates are still unavailable for Linux for particular reasons, such as DLL restrictions and other environmental issues.

Projects consolidated in the .NET Framework for desktop development do not have support for Linux, such as **Windows Presentation Foundation** (**WPF**) and Windows Forms, at least not natively. Some specific open-source projects and initiatives allow the running of WPF applications on

Linux or other applications using Electra. Still, the restrictions make any project of that nature quite risky.

Web Applications, since .NET Core 1.0, are entirely compatible with the Linux operating system. That includes Asp.Net MVC, Asp.Net Razor, and Blazor applications. It is possible to develop over any of these templates and choose to deploy the application in the Linux operating system instead of Windows. It represents a turning point in costs with the operating system and allows us to exchange between platforms depending on the scenario.

Using Visual Studio 2022, if you create a new project from scratch and filter the project templates by Linux in the platform select option, you will find the available project options for Linux, as shown in *Figure 12.1*:

Figure 12.1: *.NET project templates for Linux*

Currently, it is possible to create for Linux from Console Applications to Blazor Apps based on Web Assembly, including the most standard Asp.Net Core applications, such as Razor pages, Asp.Net MVC, Asp.Net Web API, and others. In the following section, you will understand and learn how to develop .NET applications for Linux using WSL 2.

Developing .NET applications with WSL 2

Since the first .NET Core versions, it has been possible to create .NET applications in Windows and debug the same application in Linux, verifying the application's behavior locally before deployment in an actual server. That is an important feature available for .NET developers as the software testing process for cross-platform development requires checking for overall quality in multiple operating systems.

WSL is an acronym for Windows Subsystem for Linux, allowing us to debug .NET applications in Linux without having to deploy the application in an actual Linux server. You can do the whole process in the Windows operating system and choose many Linux distributions available for WSL, with Ubuntu installed by default. This is handy for productivity and represents a powerful tool for any developer who wants to test .NET applications in Linux properly.

To start using WSL 2, you first need to install it on Windows. According to Microsoft's official documentation, this tool is available for Windows 10 version 2004 and higher versions; you can get it at the following link: **https://docs.microsoft.com/en-us/windows/wsl/install**.

Using Powershell or Windows command-line tool (cmd) in the Administrator mode, you can install WSL by running the following code:

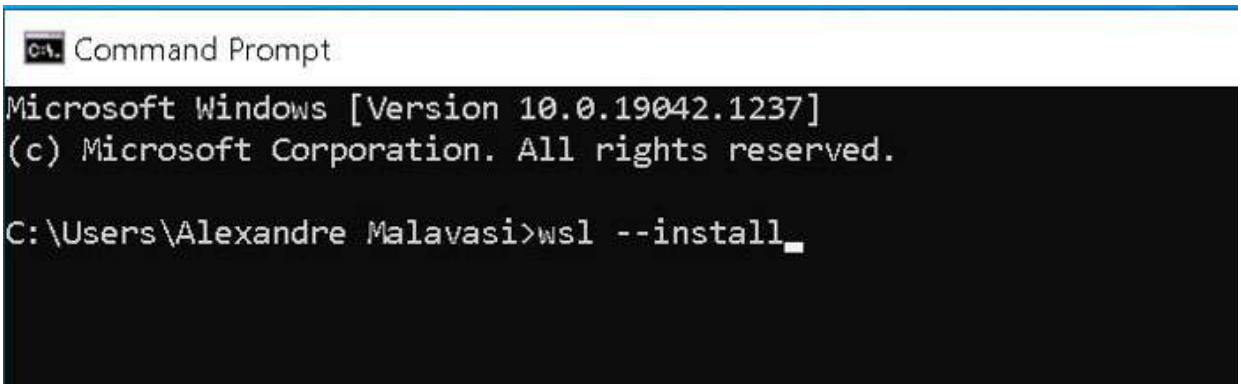


Figure 12.2: *WSL 2 installation command line*

The installation downloads all the necessary files necessary to run WSL. Choosing the Linux distribution you want to use is possible in the middle of this process. The Ubuntu distribution is installed by default if another version is not selected, as shown in *Figure 12.3*:

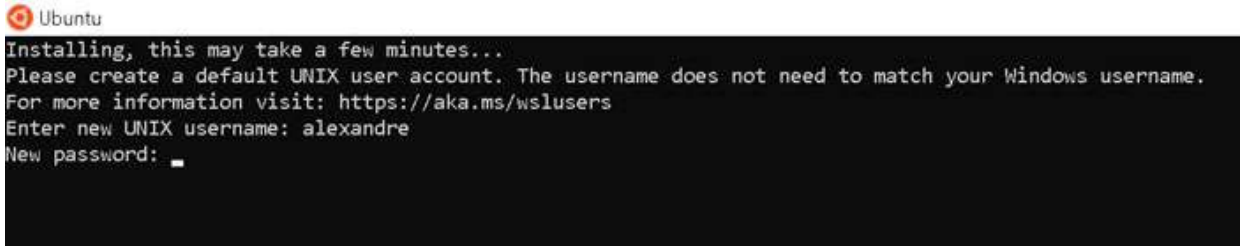


Figure 12.3: *Ubuntu installation*

The Linux installation process will require the setup of a username and password that you will need to use to integrate Visual Studio and WSL 2 later. If the installation succeeds, you will see details on the Linux distribution, as shown in *Figure 12.4*:

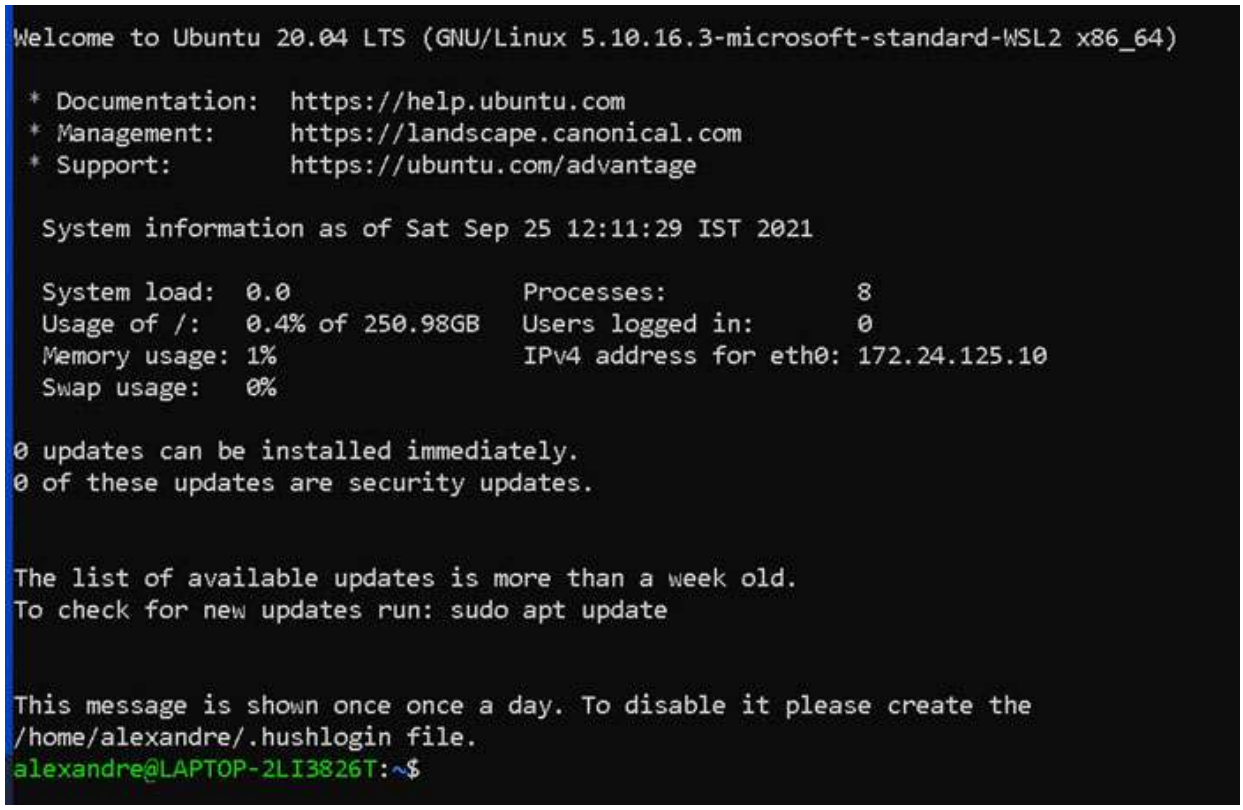


Figure 12.4: *Ubuntu details*

Visual Studio 2022 allows us to launch .NET applications directly on the WSL presented on the same machine to simulate the behavior of the applications in Linux. With previous versions of Visual Studio, such as the 2019 version, the installation of extra extensions was needed to provide the same experience that Visual Studio 2022 already provides.

After installing WSL 2 on your Windows machine, you are already able to start running .NET applications in Visual Studio targeting Linux. To verify how it works, you can create a new Asp.Net Core MVC application using Visual Studio 2022. First, create a new project, choosing the Asp.Net Core Web App template, as shown in *Figure 12.5*:

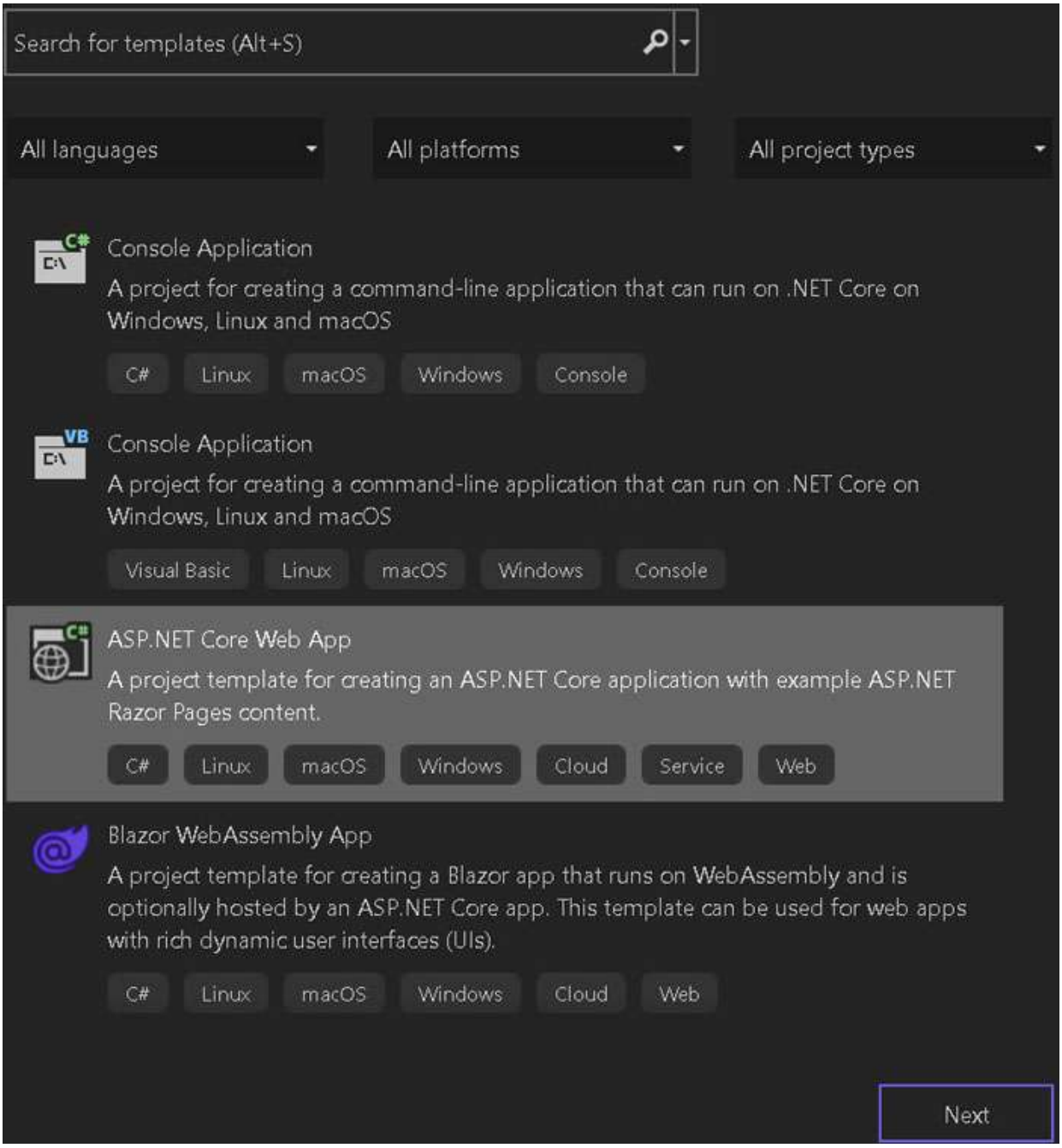


Figure 12.5: *Asp.Net Core Web App template*

After giving the project a name in the next steps of the project wizard, choose the .NET version you want (.NET 5 or .NET 6) and confirm the creation. The default template for Asp.Net Core Web App creates a Razor pages project with the structure presented in *Figure 12.6*:

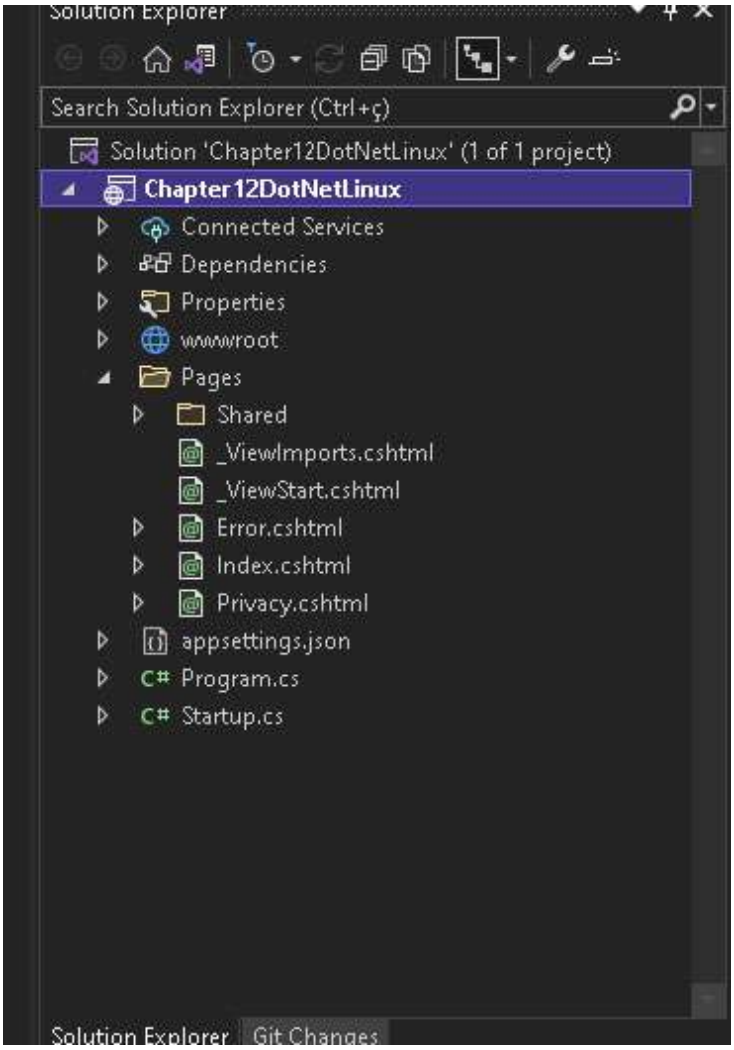


Figure 12.6: *Razor pages project structure*

In order to check the operating system that the application will be running over, in the **`Index.cshtml`** file under the Shared folder, change the content of the **`h1`** HTML tag to show the operating system, as shown in *Figure 12.7*:

```
Index.cshtml
1   @page
2   @model IndexModel
3   @{
4       ViewData["Title"] = "Home page";
5   }
6
7   <div class="text-center">
8
9
10      <h1 class="display-4">@Environment.OSVersion</h1>
11
12
13      <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">
14      building Web apps with ASP.NET Core</a>.
15
16      </p>
17  </div>
18
```

Figure 12.7: *Index page*

In *Figure 12.7*, you can see the information Environment.OSVersion is displayed through line 10 of the code. This code renders on the page the application's operating version is running on. Suppose you run the application using the default configuration on Visual Studio, as the operating system used in the machine is Windows. In that case, the underlying information is shown on the page, as shown in *Figure 12.8*:

Figure 12.8: *Windows operating system*

However, as WSL 2 is installed and configured in the same machine for Ubuntu Linux distribution, it is possible to choose the WSL option in the running mode within Visual Studio, as shown in *Figure 12.9*:

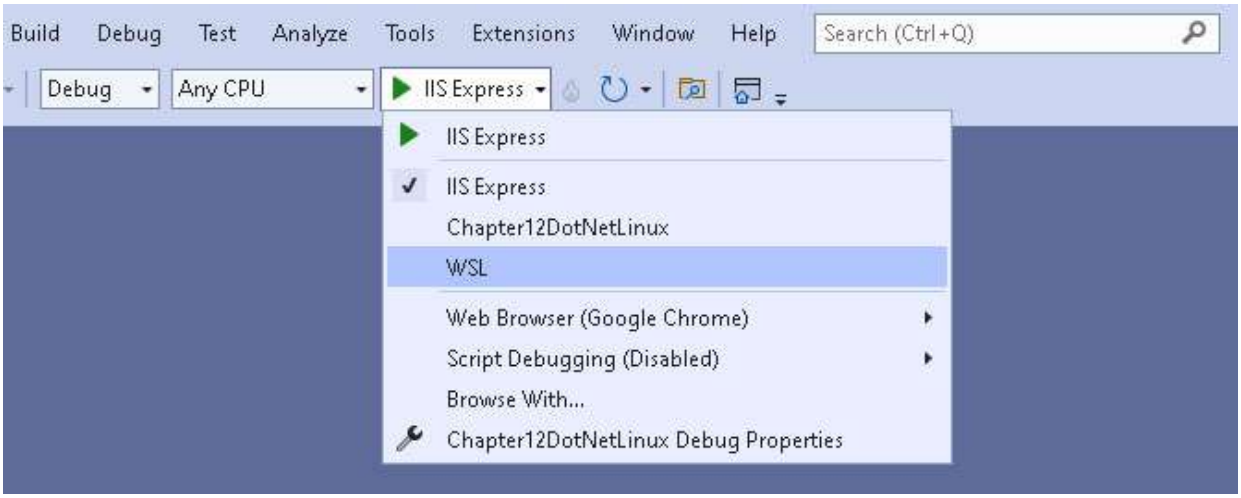


Figure 12.9: *WSL debug option*

By choosing the WSL option, Visual Studio launches the .NET application within WSL under Linux Ubuntu or under any Linux distribution that is available and configured, as shown in *Figure 12.10*:

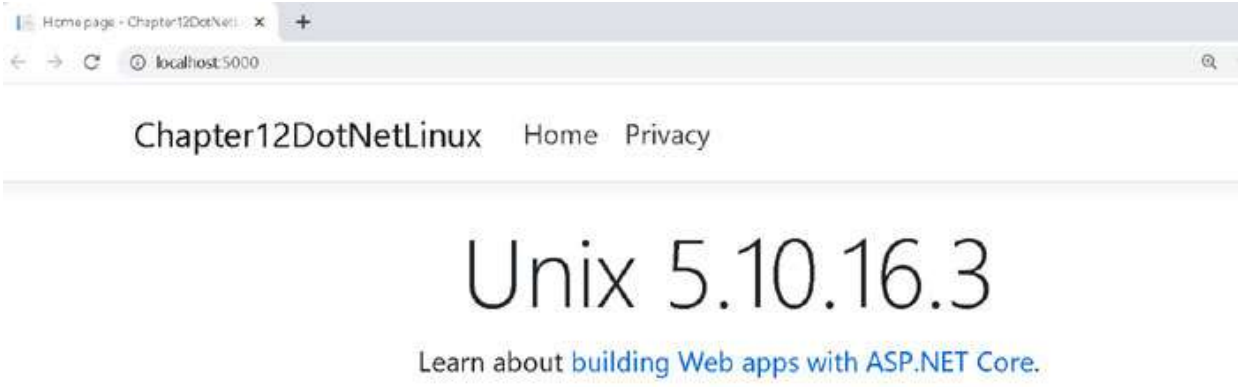


Figure 12.10: *.NET application running on Linux*

After running the application, it is possible to confirm the Environment.OSVersion statement recognized Unix 5.10.16.3 as the

operating system version on which the application is running. The same application works perfectly well on Windows and Linux, and using Visual Studio 2022 and WSL makes it possible to debug the .NET application in many Linux distributions. That is an essential feature for any developer who develops cross-platform applications. They need to ensure that the software has the expected result in multiple operating systems before any deployment to actual production environments.

In *Chapter 15, 'Desktop, Console and Mobile Applications*,' you will have more opportunities to exercise the creation of cross-platform applications, including options for Linux, with the use of .NET MAUI.

Conclusion

As seen in this chapter, .NET has many project templates that fully support the Linux operating system, including the possibility of debugging applications on different distributions of Linux, even developing in Windows through WSL. This represents a considerable advantage in cross-platform development and allows us to thoroughly test the application in more than one operating system to figure out issues before production deployments.

In this chapter, you learned Windows Subsystem for Linux and how to debug .NET applications in Linux Ubuntu using Visual Studio 2022. Furthermore, you had the chance to familiarize yourself with the main advantages of cross-platform development, such as benefits in the hiring process, server costs, and much more.

In the next chapter, you will have the chance to learn how to build Web Applications in .NET, covering Asp.Net MVC, Razor Pages, and Web APIs.

Points to remember

- WSL is the acronym for Windows Subsystem for Linux and allows us to debug .NET applications in Linux, even when developing the application in the Windows operating system.
- WSL has Ubuntu as the default Linux distribution.
- Cross-platform development is a key point as a market penetration strategy for any software company.

Multiple choice questions

1. **Which command can be used to install WSL using Powershell or CMD?**
    a. dotnet install
    b. Linux Ubuntu --install
    c. WSL dotnet install
    d. WSL --install
2. **What is the default distribution in WSL installations?**
    a. Debian
    b. Fedora
    c. Ubuntu
    d. Mandriva
3. **Which project template in .NET does not have full support for Linux?**
    a. Blazor
    b. Windows Forms
    c. Asp.Net MVC
    d. Razor Pages

Answers

1. **d**
2. **c**
3. **b**

Questions

1. Create an Asp.Net Core MVC project and change the Home Index view to show the operating system. After that, run the application in WSL.
2. List and explain the advantages of cross-platform development.

Key terms

- **WSL** is an acronym for Windows Subsystem for Linux and represents a layer on Windows compatible with executables used in Linux.
- **Ubuntu** is one of the Linux distributions created in Linux communities with the option for professional support despite the free operating system.
- **IDE** is an acronym for Integrated Development Environment and represents software used by developers to build and execute applications.
- **DLL** is an acronym for dynamic link Library and represents a group of functions and components that can be reused by other programs to execute specific tasks.
- **MVC** is one of the most used software architecture patterns in the market for Asp.Net Core applications. The acronym stands for Model-View-Controller, which explicitly says that it highlights the separation of concerns in the application in logical layers.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 13

Asp.Net Core Web API

Introduction

The .NET platform has consolidated frameworks for Web development under Asp.Net Core that meet the most high-standard requirements of the market in terms of security, performance, flexibility, and other fundamental concepts that allow us to build powerful cross-platform Web applications based on Razor syntax and APIs REST.

Learning the Asp.Net Core project types allows you to identify in which scenarios they can be used in different contexts to solve various problems regarding Web development.

This chapter will have a detailed overview of the most common uses for Asp.Net Core Web API projects, including Minimal APIs.

Structure

In this chapter, we will discuss the following topics:

- Asp.Net Core Web API project
- HTTP Verbs
- Creating a web API project
- Minimal APIs

Objectives

After studying this unit, you should be able to identify the scenarios where the Asp.Net Core Web API project can be used. You will also learn to create simple Web API projects using the .NET platform. You will understand standard concepts of Web APIs.

Asp.Net Core Web API project

With the popularity of Web-based applications over the last two decades, the need for integration between services exponentially increased, forcing the software development market to establish standard protocols that would allow communication between systems in a more generic approach. Even the different parts involved in an integration process are built using other backend technologies.

In this regard, it is crucial to understand the concept of APIs based on REST, which is the most common type of cross-platform communication for Web development. API stands for Application Programming Interface, and its primary purpose is to define a typical contract between multiple applications, establishing the pattern used for requests and responses. There are many ways to facilitate communication between different systems through APIs, but over the last years, REST has been the most used architectural standard used for Web development, which means Representational State Transfer.

In this API REST model, the technology used between the client and server does not have any restrictions and can be completely different. The service that allows inter-communication follows a generic pattern for the request and response, usually using JSON (JavaScript Object Notation) format.

The contract between two or more services that will communicate with each other defines a set of specifications for methods, endpoints, parameters, authentication, and any additional relevant information on the requests and responses that exist in a specific API. But REST is not the protocol for communication but an architectural style. Typical applications based on Web development have HTTP/HTTP2 as the base protocol for communication. Because of that, it is imperative to familiarize yourself with HTTP requests before creating any Asp.Net Core Web API application.

HTTP Verbs

When a design of any Web API takes place, a clear understanding and conscious use of the correct HTTP Verbs are fundamental as they represent an essential part of the contract established in the communication between systems. Five HTTP verbs are usually applied in common Web APIs as explained in the list below.

- **POST**: This verb is commonly used to create new objects or records through an API endpoint. If the creation of the underlying resource succeeds, the API usually returns an HTTP status 201 with the object or record that was just created. Specifying the correct HTTP verb for creating records helps you prevent security issues.
- **GET**: This HTTP verb is utilized to retrieve data through an API endpoint and usually returns JSON data under the HTTP code of 200. In some instances, a GET endpoint may return data in other convenient formats: file, plain text, XML, and others.
- **PUT**: The PUT verb is often used to update one or more records/objects, returning the HTTP status of 200 if the update succeeded. In some instances, an endpoint using this verb may create a resource instead of updating any entry, but it depends on the logical implementation presented in the endpoint.
- **PATCH**: This verb is used when only a specific part of a record is due to be updated, not the entire record. Therefore, the contract between the client and the server application states that a partial object can be sent within the request. The API will handle which fields or properties of specific records need to be updated. This is similar to the PUT verb, but the PUT is commonly used for complete updates of records, not partial ones.
- **DELETE**: As the name states, this verb is used to remove records and usually returns the code of 200 in the HTTP response if the removal is successfully done.

Regarding standard API responses and HTTP status codes, code 405 is commonly used as a response by an API if the requester does not have the authorization to use the underlying endpoint or method. Another common one is code 404, which indicates that a specific resource does not exist.

Creating a Web API project

With a better understanding of the concept of Web API based on REST and HTTP Verbs, it is time to create a Web API from scratch using Visual Studio 2022 or Visual Studio code. All the samples in this chapter use the .NET 6 version, and other examples in this book use .NET 7. Regarding

compatibility for Web APIs, the concepts demonstrated in this section do not rely on a specific version being applied in either .NET 6 or .NET 7.

Using Visual Studio 2022, create a new project using the Asp.Net Core Web API template, as shown in *Figure 13.1*:

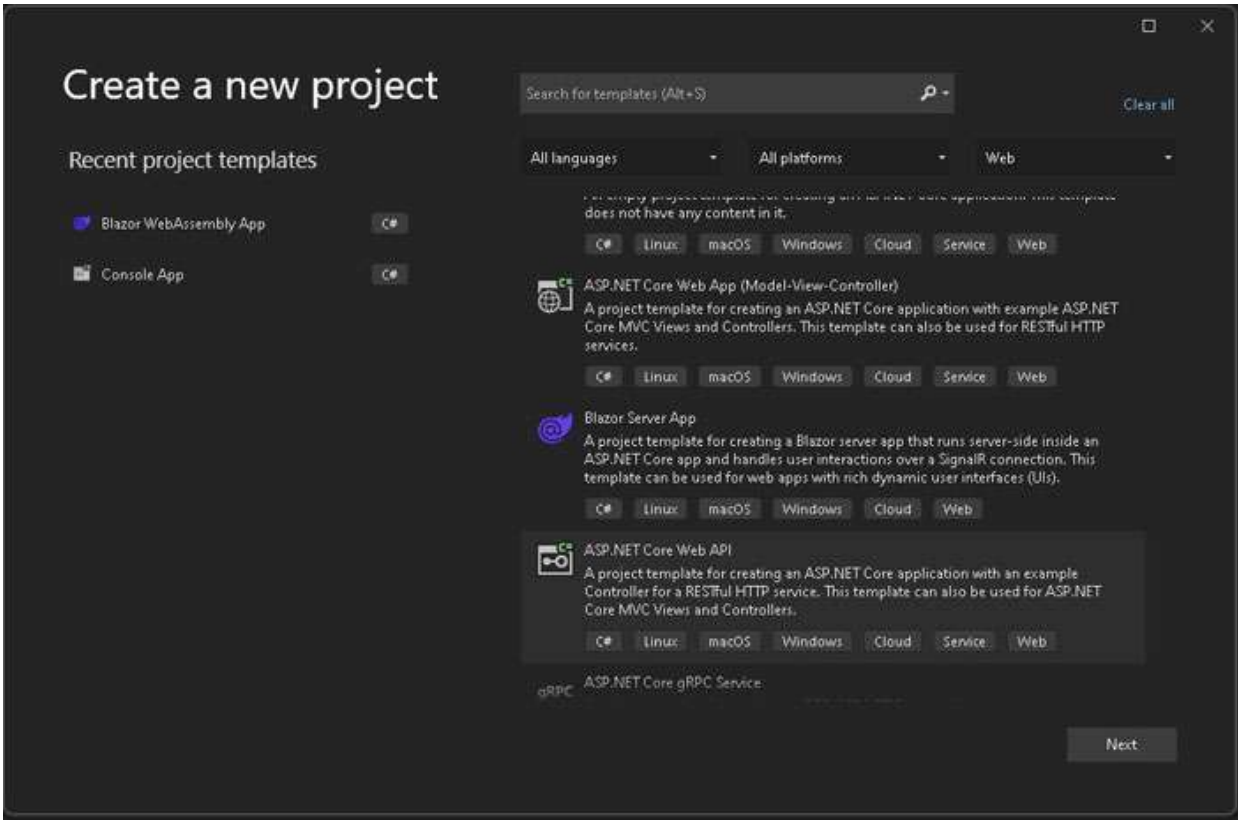


Figure 13.1: *Asp.Net Core Web API project creation*

After selecting the Asp.Net Core Web API project type, give a name and a local path for your Web API project, as shown in *Figure 13.2:*

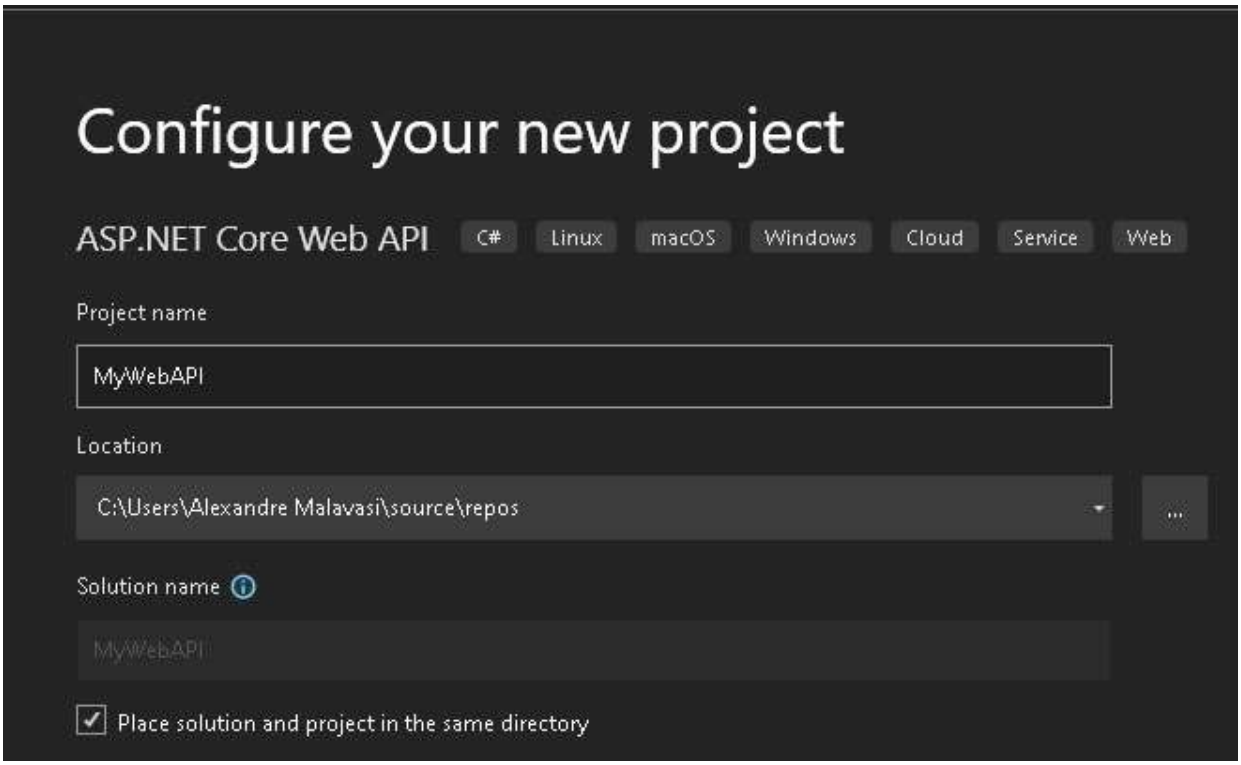


Figure 13.2: *Asp.Net Core Web API project properties*

In the next step, you will need to specify the .NET version you would like to set your Web API, the authentication type if any, and other configurations such as HTTPS, Docker, and different optional settings, as shown in *Figure 13.3*:

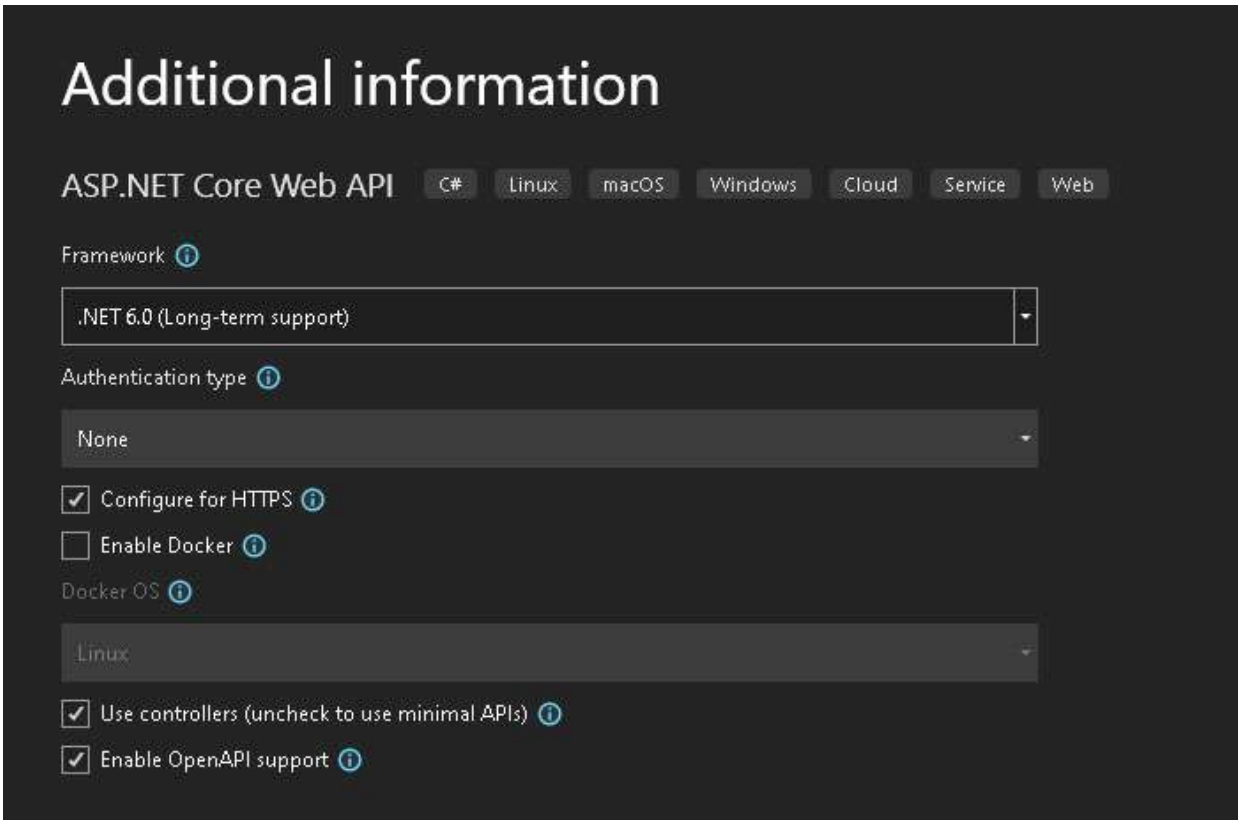


Figure 13.3: *Asp.Net Core Web API additional information*

The .NET 6 version is used for this sample project, and the configuration is set to use controllers instead of minimal APIs. The difference between controllers and minimal APIs is explained later in this chapter, including relevant examples and scenarios where each is recommended.

Once the Asp.Net Core Web API project is created, the default project template already provides a Weather Forecast Controller as an example and default settings to run the application. After you complete the project yourself, the folders and files in the Solution should be similar to the representation shown in *Figure 13.4:*

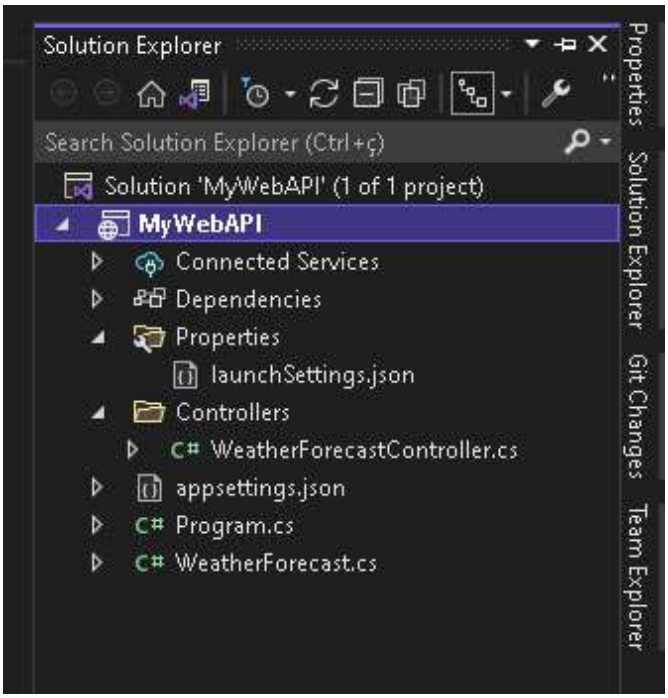


Figure 13.4: *Asp.Net Core Web API project structure*

The default project template using Visual Studio 2022 contains a single controller called **`WeatherForecastController`** and a model named as **`WeatherForecast.`** Controllers within Asp.Net Core Web API projects are responsible for intercepting the requests through endpoints, which are defined using methods combined with decorators that determine the underlying HTTP verb for the endpoint: GET, POST, PUT, PATCH, or DELETE.

If you run the Web Api project without any modification and type in the browser the URL **https://localhost:7187/WeatherForecast**, the WeatherForecast Controller intercepts the request and executes the default method in the Controller, returning a JSON result, as shown in *Figure 13.5*:

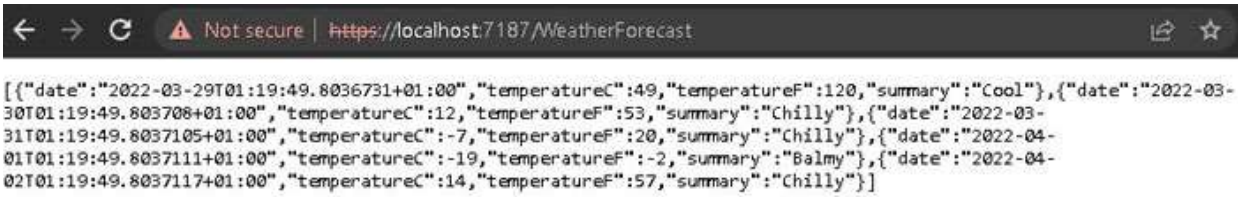


Figure 13.5: *Weather Forecast endpoint response*

The browser indicates a failure regarding **HTTPS** as the local environment does not have any valid certificate for SSL **Secure Sockets Layer** (**SSL)**; therefore, the application can still run normally. Asp.Net Core projects, in general, give us the ability to configure routes, dependency injection, authorization, authentication, and many other important aspects and features that are common in standard Web applications with the requirement for HTTPS being one of them.

Considering the default template for Web API already has something implemented, it is worth building a new controller and a new model to understand how this type of project works and can be used properly. Within Visual Studio 2022, create a new Customer class at the root of your Project Solution. The creation can be done by right clicking on the project in the Solution Explorer window, as shown in *Figure 13.6*.

The Customer class can have basic hypothetical properties, such as ID, first name, surname, birthday, and address. The underlying type of each property can be seen in *Figure 13.7*. This model does not require any methods in this example, as the entire logic is handled within the Controller. It is recommended to keep models away from logic implementation in real scenarios, with them being just Data Transfer Objects. The reason for that is to avoid dependencies between the business models and the highest layer in the application, which is, in this context, the response given by the controller as shown in *Figure 13.6*:

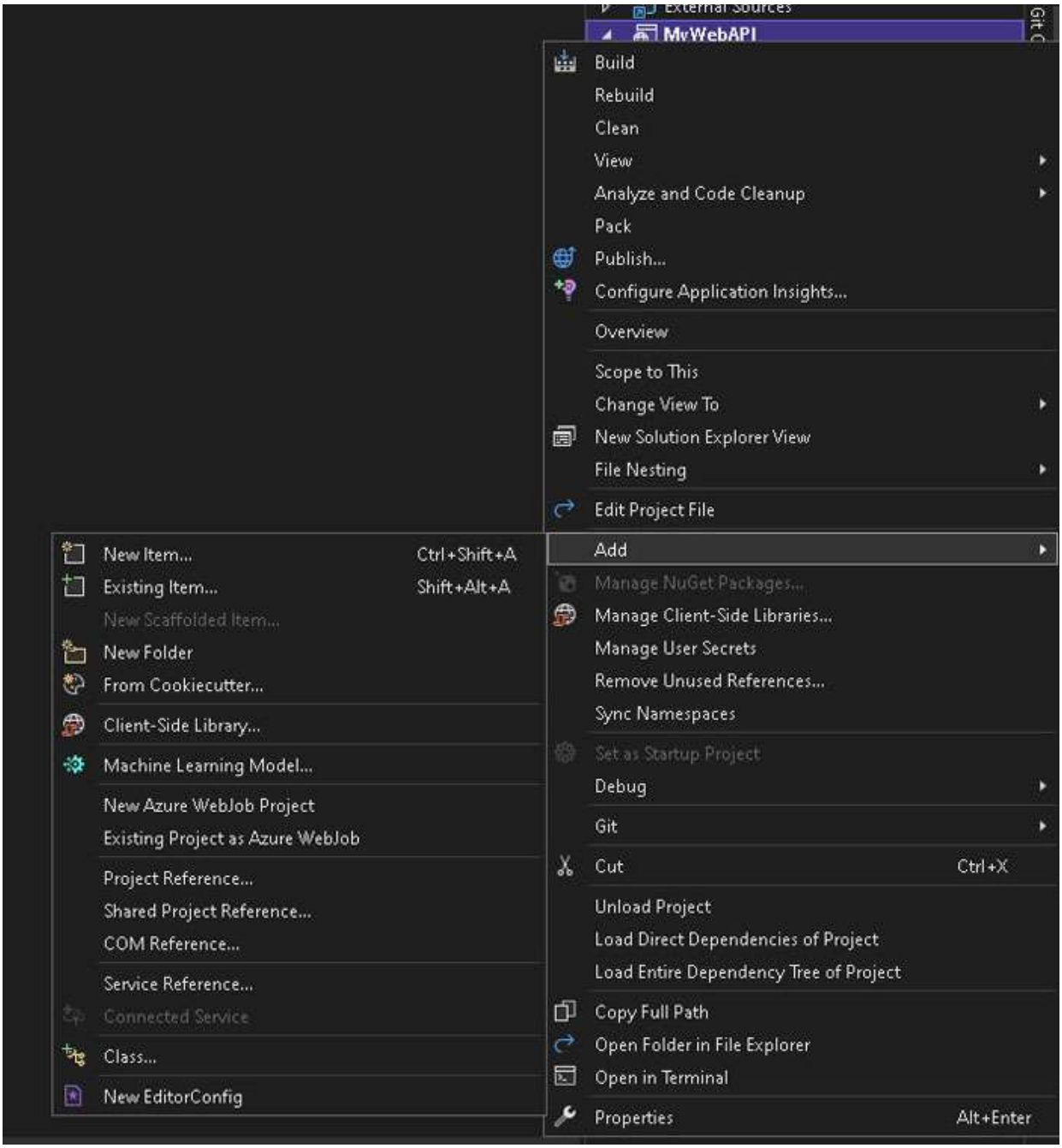


Figure 13.6: *Creation of Customer class*

After creating the Customer model, the structure looks like the one presented in Figure 13.7. The class does not require any logic on its constructor and does not have any validation for any of the properties. A proper validation would be desirable and implemented in a suitable Business layer to avoid issues for POST, PUT, and PATCH methods within the Controller, as these types of endpoints are related to data changes. Please refer to *Figure 13.7*:

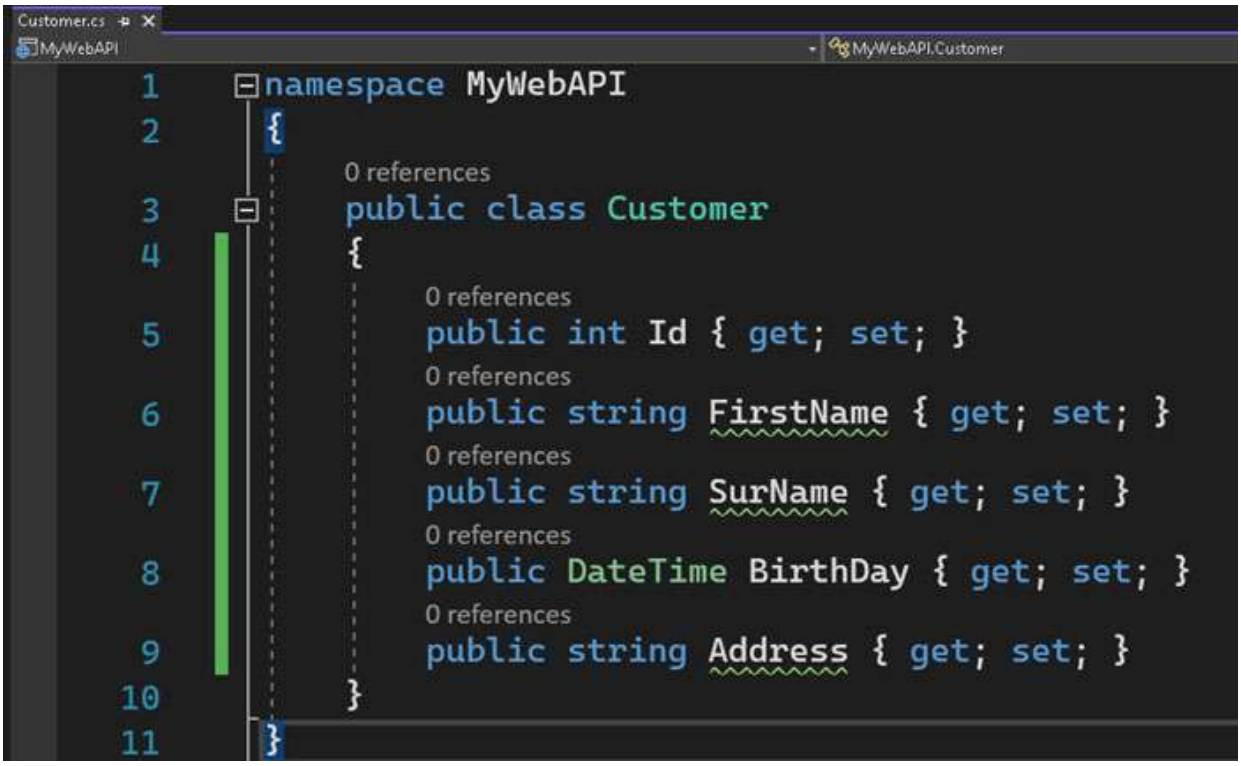


Figure 13.7: *Customer class structure*

The next step is to create the **Customer Controller**, which will have four primary operations: create a customer, update a customer, get a customer, and delete a customer. Considering the purpose of this section is to demonstrate how Asp.Net Core Web APIs work, connections with databases and more complex business logic are intentionally hidden to facilitate comprehension for the reader. Within the Solution Explorer in Visual Studio, right click on the Controllers folder, as shown in *Figure 13.8*.

Choose the "**`MVC Controller – Empty`**" option and give it the correct name as **`CustomerController`**. By convention, all the Controllers within Asp.Net Core projects have the word "Controller" as a suffix in the name of the class. This pattern facilitates the definition of automatic routes for endpoints and consistency between classes of the same type.

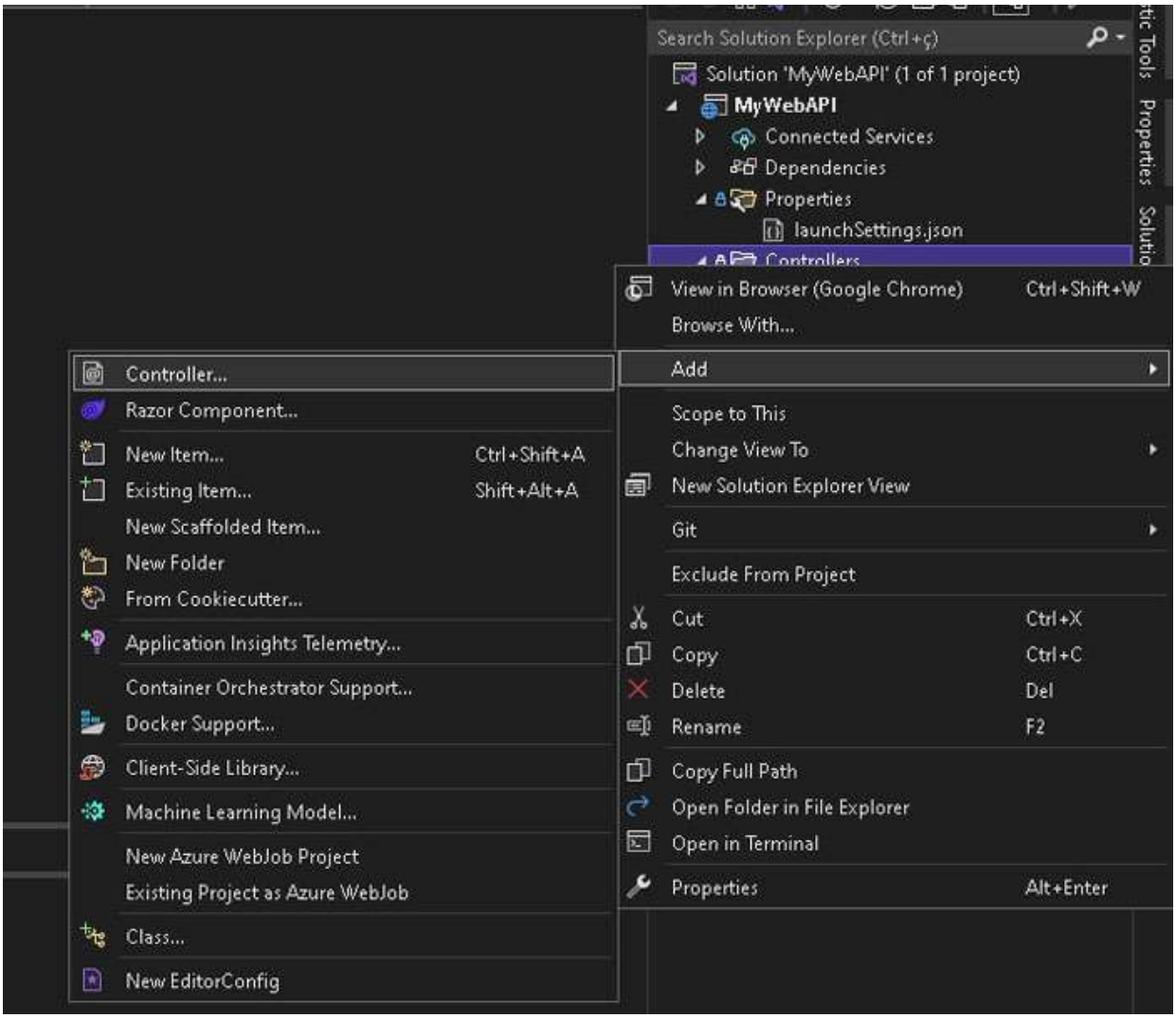


Figure 13.8: *Customer Controller creation*

After the creation of the Customer Controller, the default content looks like the representation shown in *Figure 13.9*:

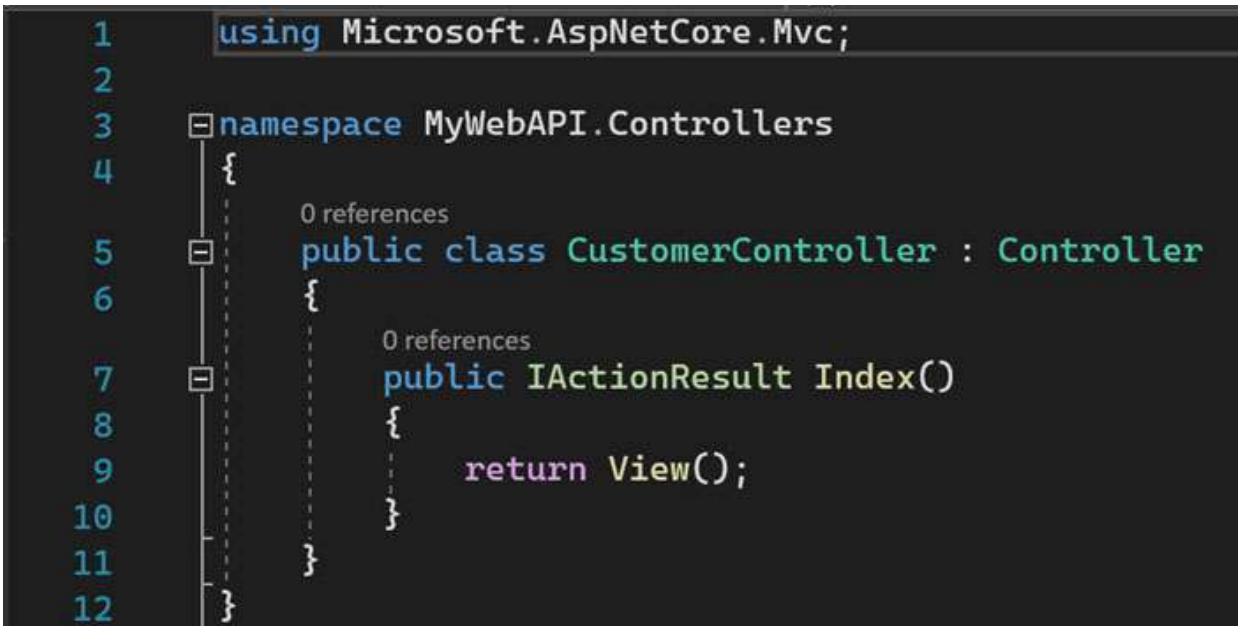


Figure 13.9: *Customer Controller default content*

The Customer Controller class inherits from the Controller class, natively presented on. NET. This configuration is automatically added when a new Controller is created using Visual Studio. However, in the context of the example of this section, the Customer Controller can inherit from the **`ControllerBase`** class to keep consistency with another Controller presented in the Solution.

Routes are a crucial part of the configuration of any Controller. Routes are the paths that expose the Controller methods as valid endpoints to be consumed by other applications. In the example of this section, the Controller's name can be part of the path, and each method complements the full path. Therefore, the endpoints have the following path: /controller/method.

Routes can be configured in class and method levels using proper Annotations. For the Customer Controller, the route for the class has the route configuration highlighted in *Figure 13.10*:

Figure 13.10: *Customer controller route*

In *Figure 13.10*, note that changes are made into Customer Controller, and it passes to inherit from **`ControllerBase`** class and returns an "Ok" response instead of a View. In line 5, an Annotation name **`ApiController`** is used to mark that class as an actual Web API. Additionally, an Attribute Route is used in line 4 to define that the Controller's name will be the root path to access all the endpoints within the underlying Controller.

A static property is used in this example to simulate the connection to an actual database. Therefore, the state of the content is kept in the memory during the time the application is running, as presented in *Figure 13.11*:

```
0 references
public class CustomerController : ControllerBase
{
    private static List<Customer> _customers = new List<Customer>()
    {
        new Customer()
        {
            Id = 1,
            FirstName = "Alex",
            SurName = "Malavasi",
            BirthDay = new DateTime(1987, 12, 1)
        },
        new Customer()
        {
            Id = 2,
            FirstName = "Myla",
            SurName = "Belle",
            BirthDay = new DateTime(1993, 5, 23)
        },
        new Customer()
        {
            Id = 3,
            FirstName = "Richard",
            SurName = "Lintz",
            BirthDay = new DateTime(1965, 4, 10)
        }
    };
    0 references
    public IActionResult Index()
    {
        return Ok();
    }
```

Figure 13.11: *Static customer list property*

As shown in *Figure 13.11*, the private property called "**`_customers`**" is a list of customers with three initial items. With this initial set, the methods for the Customer Controller can be created to add new entries to that list, update a specific item, or even remove a customer from the list. These operations would likely be done using a database or table storage system to store records in a real system.

The first method that can be created is the one responsible for getting the list of all customers in the list under the GET HTTP verb. To achieve that, create a new method within the Controller, and specify the HTTP verb and the route as attributes in the method level, as shown in *Figure 13.12*:

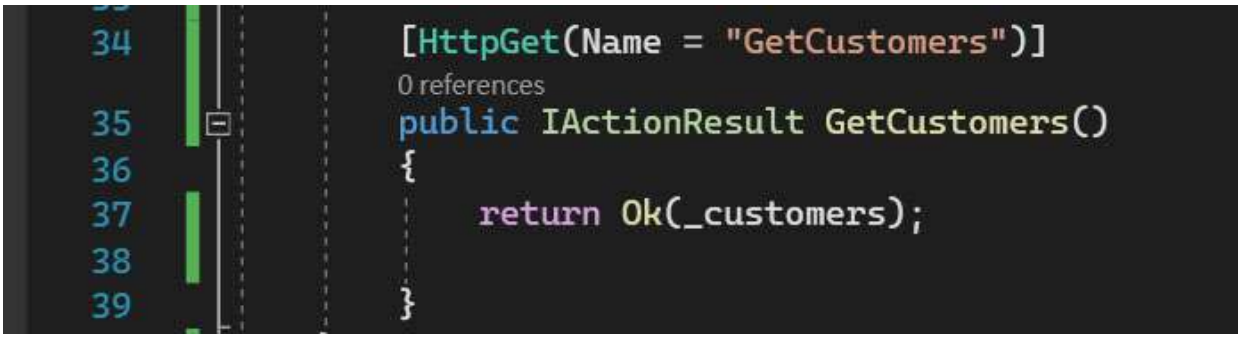


Figure 13.12: *Get Customers method*

As seen in *Figure 13.12*, the Get Customers method returns an "Ok" response with the list of customers as an argument. The "Ok" method returns an HTTP status of 200, which means a successful response for standard Web APIs. In line 34, an HTTP verb is specified, determining that this method only allows GET requests. Additionally, the Attribute above the method sets the route name for the method, which is '**`GetCustomers`**'.

If you run the application, you should be able to execute the underlying URL for the customer controller, followed by the method name: **`/Customer/GetCustomers`**. To have this path functioning with this exact configuration, go to the beginning of the Controller and change the Attribute for the whole class to include the "action" in the path, as highlighted in *Figure 13.13*:

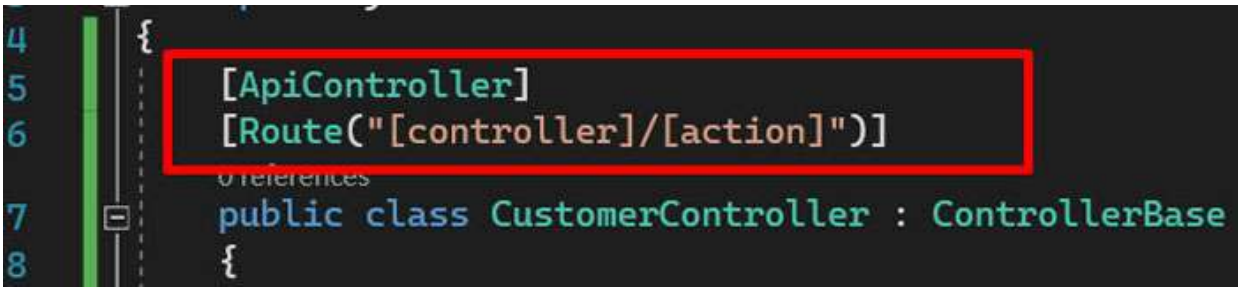


Figure 13.13: *Action configuration*

This route configuration states that the path for all the endpoints within this Controller is the name of the actual Controller, followed by the name of the method being executed. Asp.Net Core projects allow us to configure routes with broader possibilities, including dynamic routes or even query parameters.

After running the application and executing the URL **`/Customer/GetCustomers`**, the Web API executes the Controller and

methods corresponding to the route configuration. Considering the **`GetCustomers`** method returns the list of customers, *Figure 13.14* shows what the response looks like:

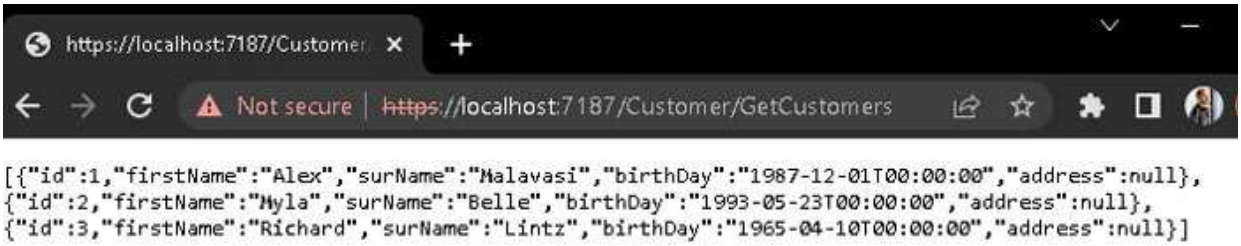


Figure 13.14: *Get Customers response*

The Web API returns a JSON object by default, serializing the List of Customers into this format. This is the most common response used by Web APIs, not only by projects based on the .NET platform. This format is primarily compatible with many technologies, including front-end applications based on JavaScript frameworks.

The next step in our example is to create another GET method that brings a single customer by receiving the customer's ID as a parameter. To achieve that, create a new method called "**`GetCustomer`**", with an integer parameter called "**`Id`**", as shown in *Figure 13.15*:

```
41      [HttpGet(Name = "GetCustomer")]
        0 references
42      public IActionResult GetCustomer(int id)
43      {
44          var customer = _customers
45              .FirstOrDefault(x => x.Id == id);
46
47          if (customer == null) return NotFound();
48
49          return Ok(customer);
50
51      }
```

Figure 13.15: *Get Customer method*

In the Get Customer method, an integer parameter is specified to intercept the customer's ID passed through the query string in the request. For example, the method body tries to get the customer from the static customer list. If the ID passed is invalid, the API method returns a “NotFound” response, representing the HTTP response status of 404. The customer object is returned as a JSON object if the customer does exist.

After running the application, you should be able to execute the new Get Customer endpoint, passing one of the valid IDs presented on the static customer list, as shown in *Figure 13.16*:

Figure 13.16: *Get Customer method response*

As ID number one is passed through the URL parameter, the Controller method intercepts the value in the request. It executes the logic successfully, responding to an HTTP status of 200 with the underlying customer, whose ID is number one. Considering the method had a logic to return a “**`Not Found`**” response for an invalid ID, as shown in *Figure 13.17*:

Figure 13.17: *Not Found response*

Passing a value of “222” as a URL parameter, the API returns a “Not Found” response with the status code of 404, as expected. Other status codes can be used for various situations, such as authorization, server error, and so on. Each type of response helps the client application properly handle errors between requests and track the root cause for issues in integrations. Therefore, while building your API in real scenarios, specify the correct

response based on standard practices used in the market for business applications.

As we already have two GET methods in the Customer Controller, the next step is to create suitable methods to add and update customers to the static list. Starting from the Add method, create a new Controller action that receives a customer object as a parameter, adds the new customer to the static customer list and returns the latest record using the "Ok" method. After doing that, your method would like the representation in *Figure 13.18*:

Figure 13.18: *Add Customer method*

Note that the "**`AddCustomer`**" method is configured to be a POST method in line 53 and expects to receive a customer object in the request. Considering this method allows only the POST HTTP verb, it is impossible to test it by passing parameters via URL. The customer information must be given in the header of the request.

To facilitate the test of APIs, Asp.Net Core Web APIs projects, since version .NET 6, configure a Swagger endpoint by default to document the API endpoints and provide a playground to test all the methods. To ensure you have Swagger configured correctly in your project, go to the Program.cs file within the project Solution, and check whether Swagger is specified there, as shown in *Figure 13.19*:

```
12      // Configure the HTTP request pipeline.
13      if (app.Environment.IsDevelopment())
14      {
15          app.UseSwagger();
16          app.UseSwaggerUI();
17      }
```

Figure 13.19: *Swagger configuration*

Swagger provides a documentation page built dynamically, listing all the Controllers and their methods presented in the Asp.Net Core Web API project, as shown in *Figure 13.20*:

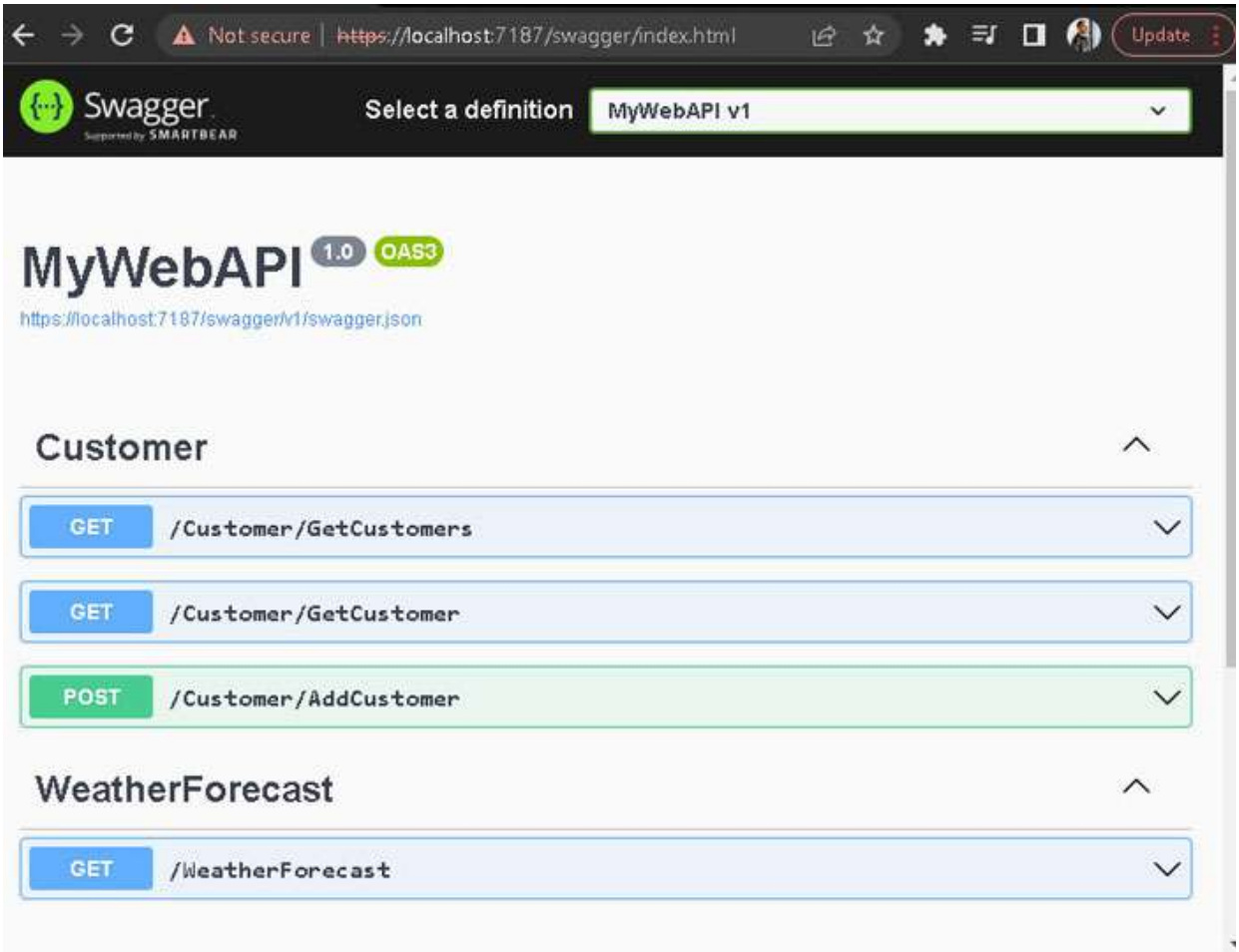


Figure 13.20: *Swagger page*

Typically, the Swagger page can be accessed through the URL "**`/swagger`**", which directs automatically to the list of endpoints available in the Web API.

To test the “AddCustomer” POST method, you can expand the underlying method on Swagger and click on the “**`Try it out`**” button, as shown in *Figure 13.21*:

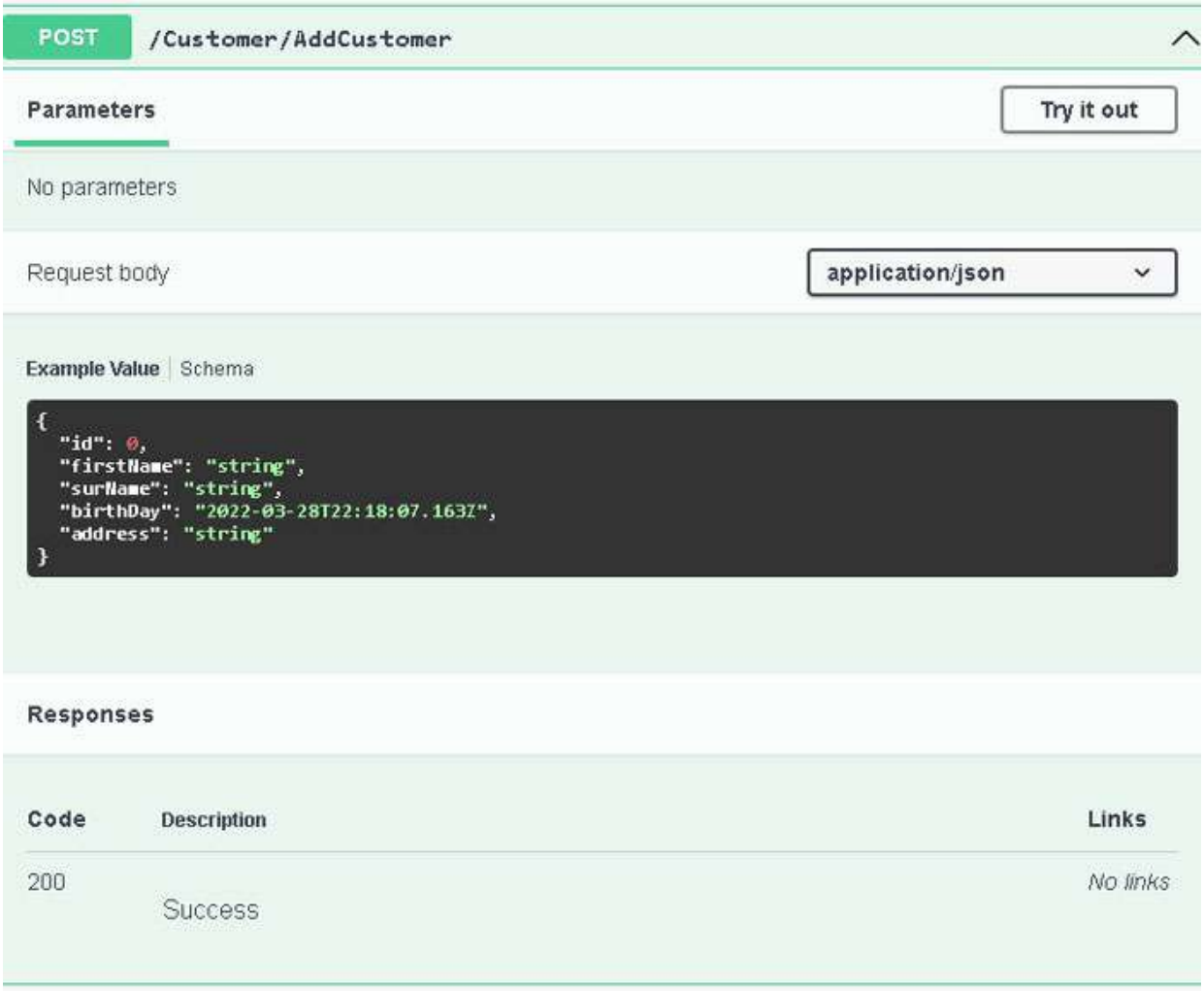


Figure 13.21: *Add Customer method on Swagger*

The “**`Try it out`**” option opens details on the necessary API request and suggests a test customer object that needs to be specified by you to test the request. After specifying random test values on all the properties in the JSON object, use the option “**`Execute`**”. This operation submits a request to the underlying method within the Customer Controller and receives the details on the API response, as shown in *Figure 13.22:*

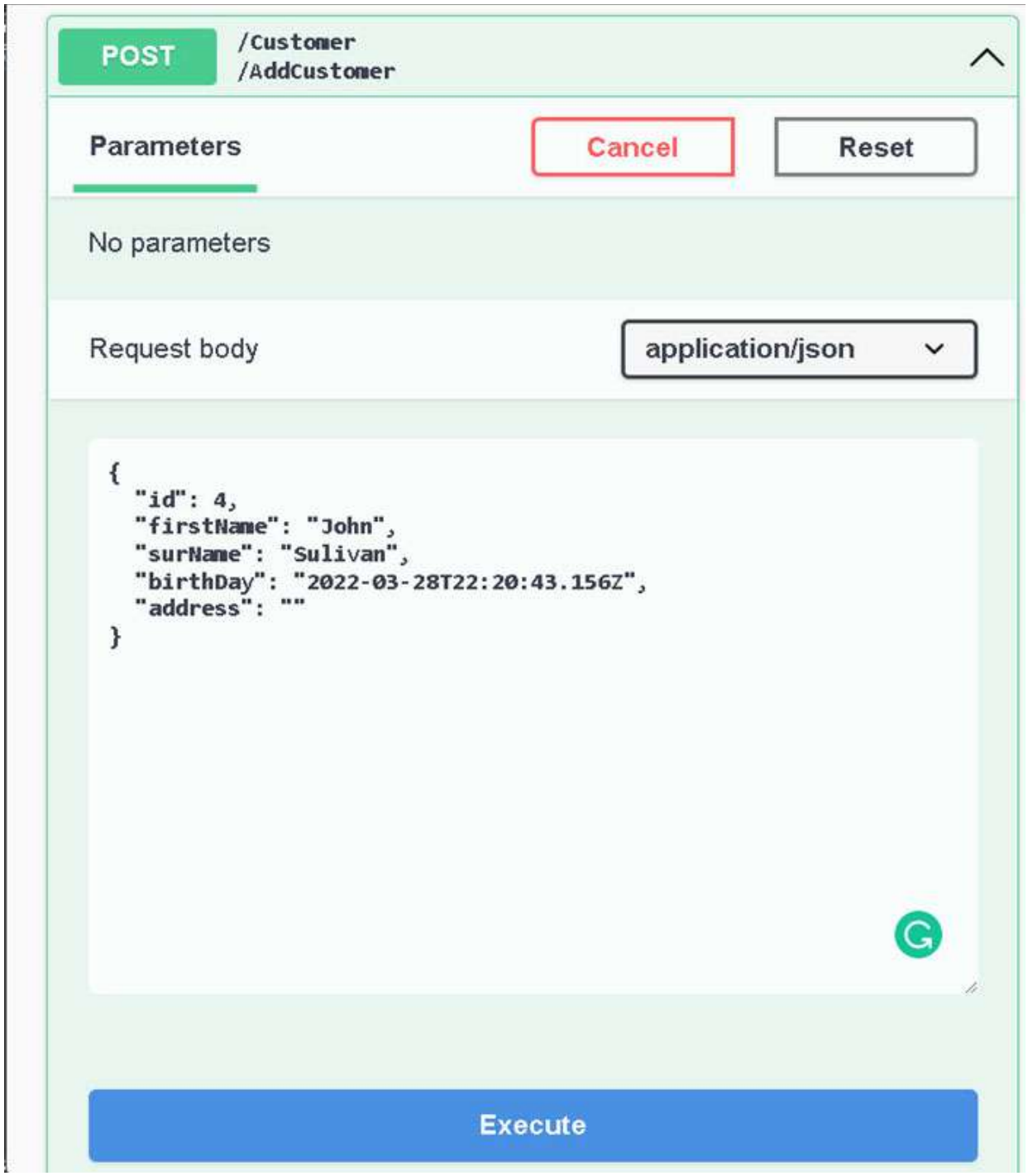


Figure 13.22: *Add Customer method execution*

After pressing the Execute button with a valid JSON customer object filled, you should be able to see the response details returned by the underlying API Controller action, with a status code of 200 and the new customer information on the response body box, as shown in *Figure 13.23:*

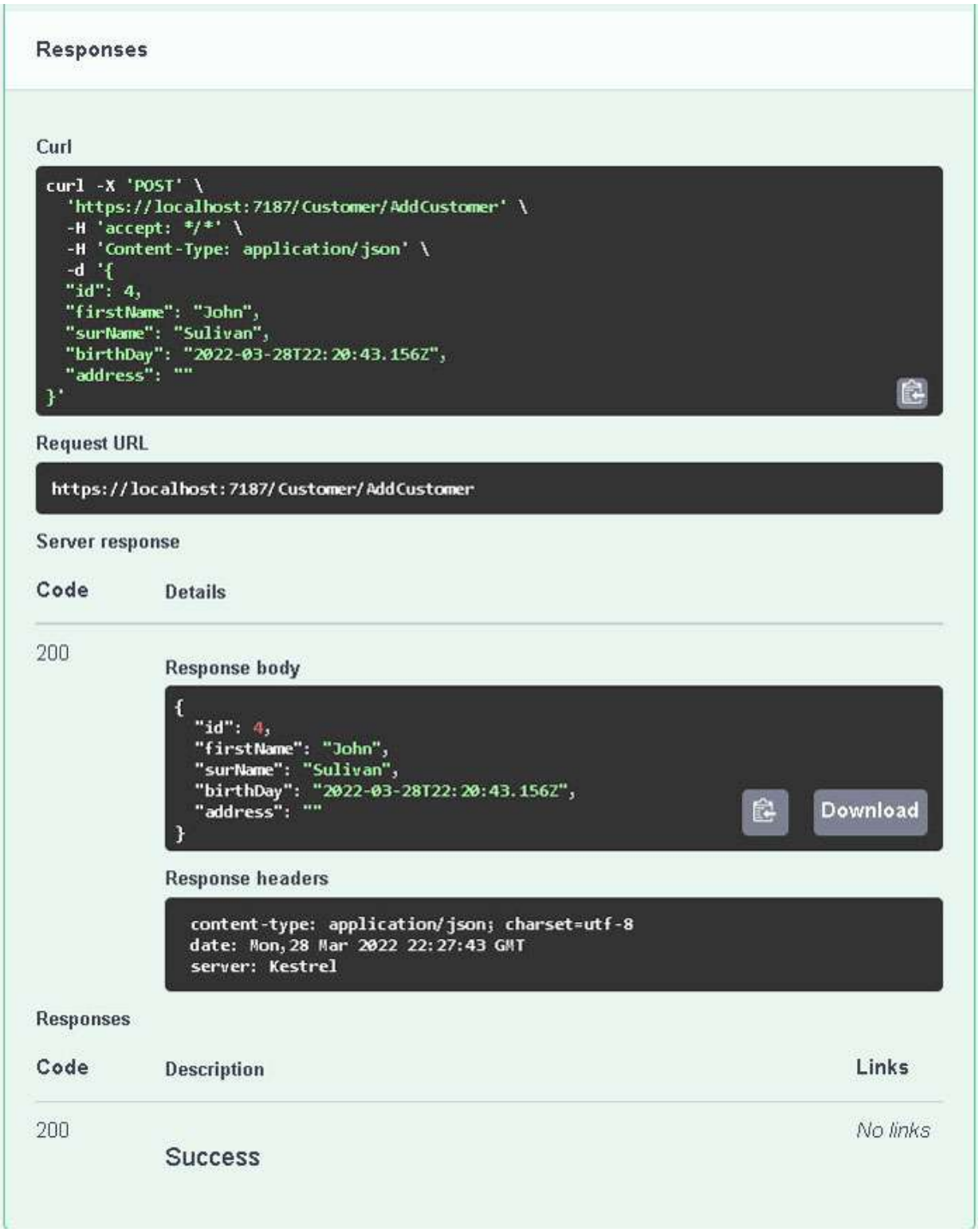


Figure 13.23: *Add Customer method response*

The same principle applies to PATCH, DELETE, and PUT HTTP verbs. Once a specific method is created within the Controller, you must specify the

underlying HTTP request type in the method attributes and implement the necessary logic for each method.

Minimal APIs

Following the same pattern used in other platforms, such as python, PHP, and others, the .NET platforms provided a way to build minimal APIs since .NET 6, being possible to create simple APIs without so much effort in terms of configuration and coding. A traditional Asp.Net Core Web API project contains many folders and files to function correctly, such as Controllers, Models, Program configuration, and so on. The concept of minimal APIs in .NET allows quickly creating a Web service with one or multiple endpoints, with a minimum of code apart from the actual logical implementation, particular to each API project.

In Visual Studio 2022, create a new Asp.Net Core Web API project, unchecking the “Use controllers” box, as shown in *Figure 13.24*:

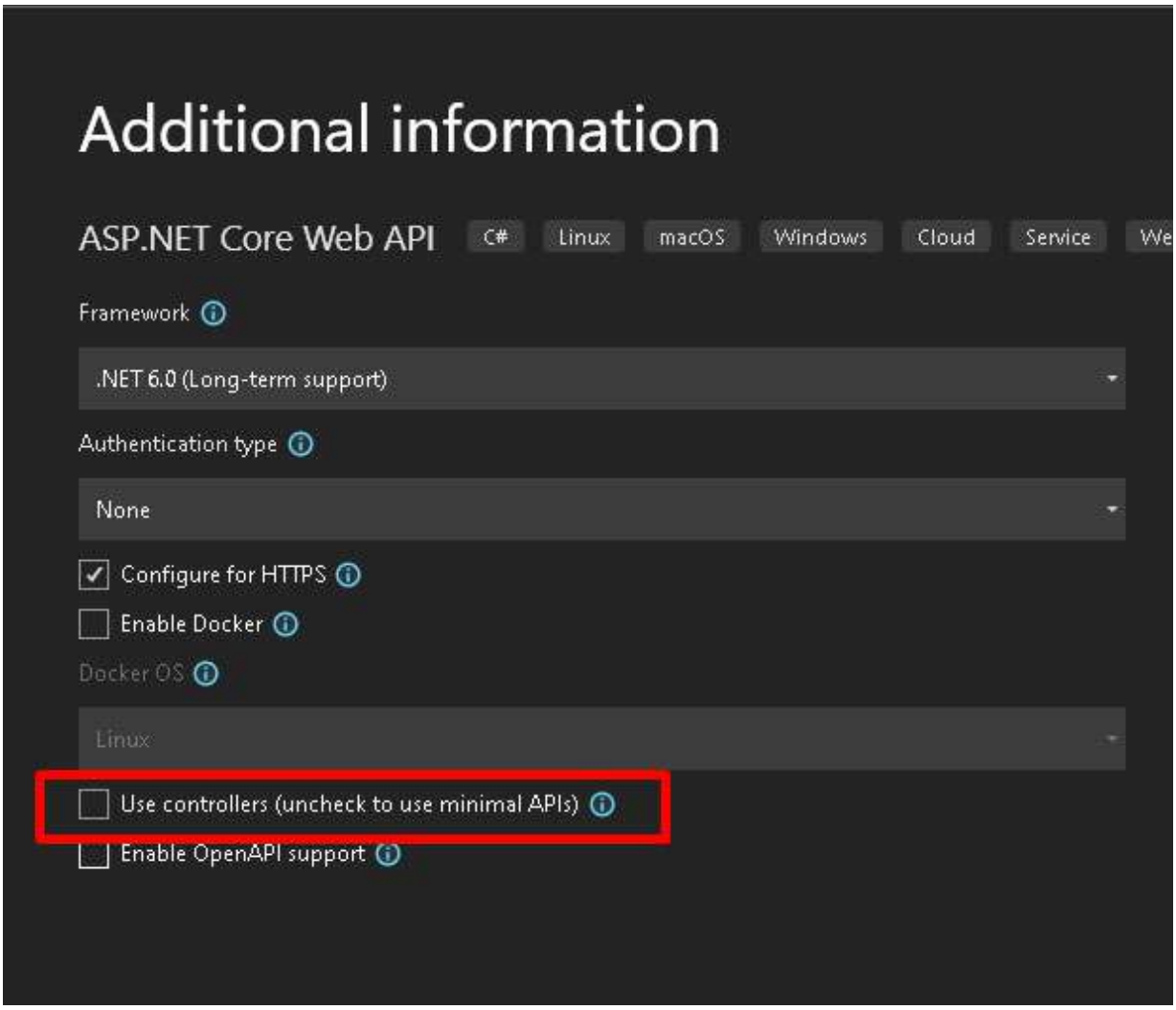


Figure 13.24: *Creation of Minimal API project*

After confirming the creation of the project on Visual Studio, it is possible to see that the number of files is much less significant than the previous Web API project created previously in this chapter. It has only two configuration files in JSON format to store the application's settings and the Program.cs file to contain the actual implementation of the minimal API, as shown in *Figure 13.25*:

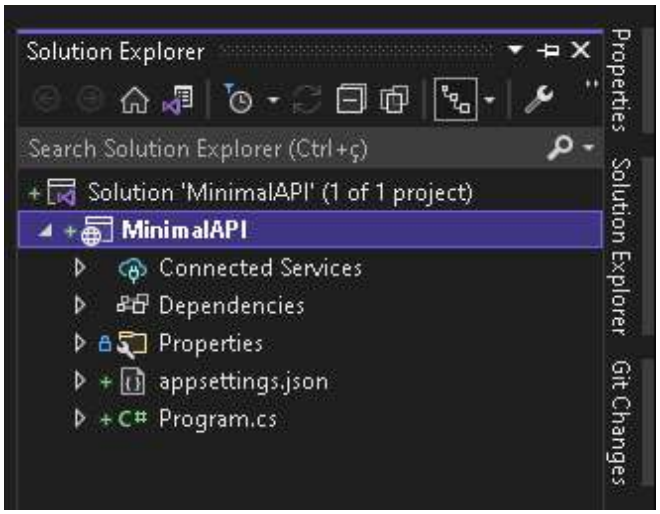


Figure 13.25: *Solution Explorer for Minimal APIs*

The Program.cs file is where you can specify the configuration for the entire application and specify the API endpoints simultaneously in a single structure. Minimal APIs projects allow you to have a similar experience to a traditional Web API project that uses Controllers and actions. The route configuration, authentication aspects, authorization, and any other configuration can be done directly in the Program.cs file, including specification of internal classes and much more, as shown in *Figure 13.26*:

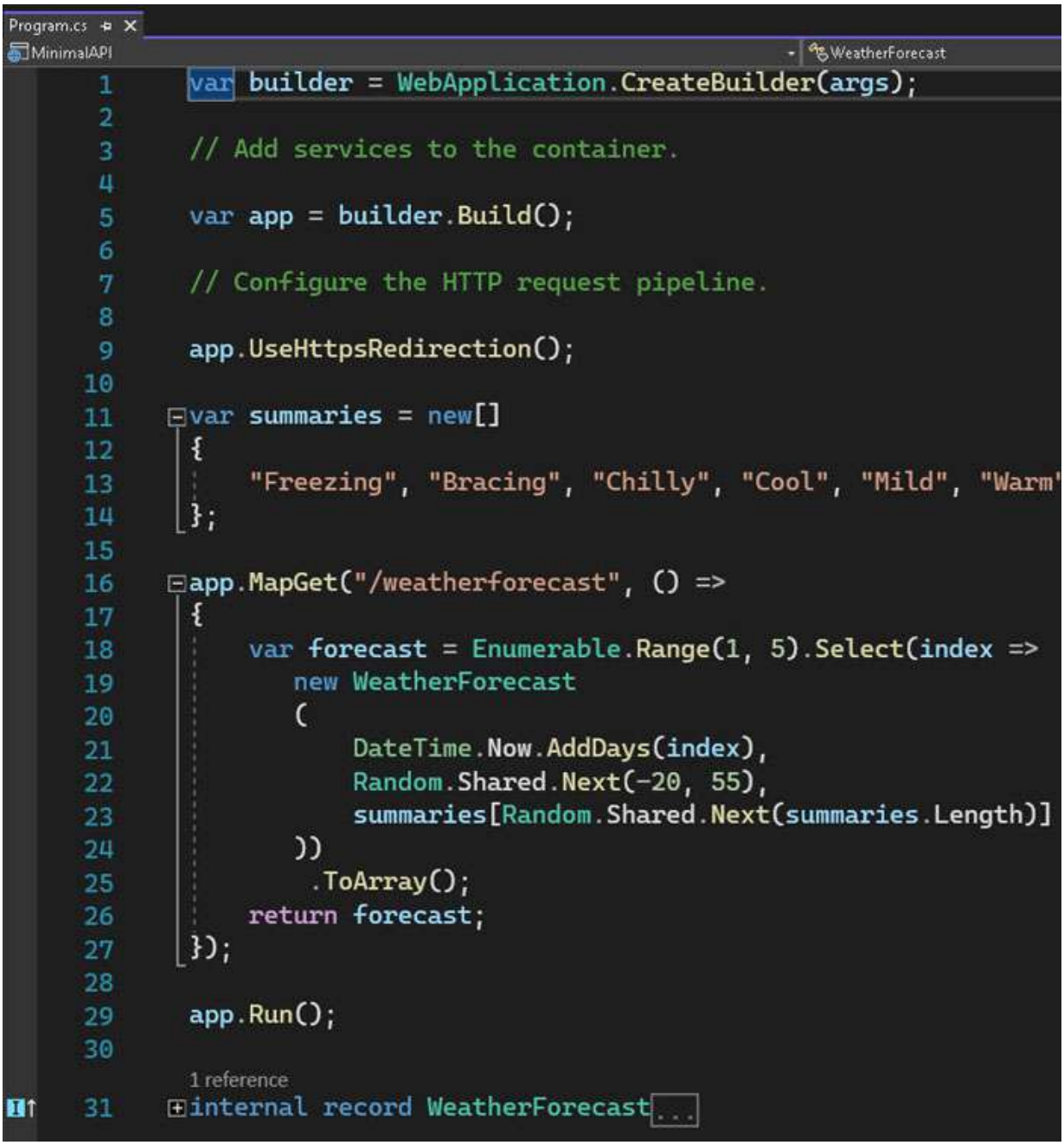


Figure 13.26: *Program.cs sample of Minimal APIs*

The default project template for Minimal APIs presented on Visual Studio contains the same Weather Forecast endpoint that exists in the project template for Web APIs based in Controllers. The HTTPS configuration and the definition of a Web API method are under the “weatherforecast” route in the same file.

The advantage of minimal APIs is that you can create simple APIs as microservices without relying on all the heavy configurations presented in traditional Web APIs that use Controllers. However, if your application in real scenarios has way too many endpoints, the use of minimal APIs is not recommended, as all the implementation is presented in the same file. In minimal APIs is not possible to split business areas into multiple Controllers. Having said that, the evaluation of its use should be carefully done.

In the next chapter, you will have the opportunity to configure authentication for Asp.Net Core projects, considering that the exact mechanism for authorization and authentication is applicable for different Asp.Net Core projects, including Razor Pages, Model-View-Controllers, and Blazor applications.

Conclusion

In this chapter, you learned that the .NET platform provides project templates to build Web API projects efficiently with intuitive use of attributes to configure routes, authentication, HTTP Verbs, and other essential properties for Web APIs.

Web APIs are essential for any business project, as they allow us to prepare applications to communicate with others without exposing the technology used in the backend. Additionally, minimal APIs are based in Asp.Net Core since .NET 6 represents a powerful and efficient way to stand up a Web Service, following similar practices used by other technologies such as Python, Go, and others.

In this chapter, you learned how to create simple Web APIs, the concept of HTTP requests, HTTP verbs, HTTP status codes, minimal APIs, and much more.

In the next chapter, you will have the opportunity to dive deeply into Blazor, the new Single Page Application of the .NET platform. You will have the opportunity to configure authentication for Asp.Net Core projects, considering that the exact mechanism for authorization and authentication is applicable for different Asp.Net Core projects, including Razor Pages, Model-View-Controllers, and Blazor applications.

Points to remember

- The use of minimal APIs should be considered for small applications.
- Specifying the correct HTTP verb while creating any API endpoint is essential to facilitate integrations between different systems.
- Swagger can be used to document and test APIs.
- Minimal APIs are presented only since the .NET 6 version.

Multiple-choice questions

1. **According to what you have learned and the practical example in this chapter, which is the HTTP verb commonly used to create new records?**
    a. PUT
    b. PATCH
    c. GET
    d. POST
2. **Which method returns a status code of 200 from a Controller method?**
    a. View
    b. Route
    c. Ok
    d. Not Found

Answers

1. **d**
2. **c**

Questions

1. Explain the reason for HTTP verbs in APIs.

2. Using the Customer API demonstrated in this chapter, create a similar version of the same endpoints using Minimal APIs.
3. Explain in which type of scenarios the Minimal APIs would be suitable.
4. What is the most common format used in API response?

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 14

Blazor, the Single Page Application of .NET

Introduction

Single Page Application became one of the most popular patterns used to build modern and high-performance Web applications, traditionally using JavaScript as the primary language. The .NET platform has introduced its Single Page Application being possible to develop powerful applications using C# as the primary language, combined with WebAssembly.

Learning Blazor will allow you to understand how to run C# language directly in the browser, reuse components, and take advantage of the Razor syntax used in other Asp.Net Core projects.

This chapter will have a detailed overview of this new Web framework for .NET applications, including aspects of authentication, component inheritance, forms, and much more.

Structure

In this chapter, we will `discuss` the following topics:

- Concepts of Single Page Applications
- Difference between Blazor Server and Blazor WebAssembly
- Razor components and data binding
- JavaScript Interop

Objectives

After studying this unit, you should be able to understand the difference between Blazor `Server` and Blazor WebAssembly, create basic Web applications using Blazor WebAssembly, and comprehend concepts of Single Page Applications.

Concepts of single-page applications

Over the last twenty years, the market for web development has drastically changed due to the high demand and complexity inherent in enterprise applications and the number of users that a worldwide application usually needs to deal with. A standard architecture for a Web application, at the beginning of the Web, consisted of browser requests to a static content compound by HTML and images in the response, as represented in *Figure 14.1*:

Figure 14.1: *Traditional Web architecture*

In this type of Web architecture, once a request is made through a standard Web browser, the Web server processes the request and returns the entire Web page as a response, including all the related files: HTML, images, videos, style, JavaScript files, and others. The exact same process happens every time the user wants to navigate to another page within the same website. The majority of the content on the Web page is quite similar: header, footer, menu, and other content are shared across multiple pages. This situation is not ideal for performance, as a considerable amount of networking and bandwidth is used in all the requests, even though the content between pages is similar.

This model has many disadvantages, such as the high costs of keeping the server resources, limitations in terms of scalability, lousy user experience, and high latency for each user request. Due to all these limitations, over the last 15 years, Web development models have been proposed to give more efficiency to Web requests, lower to overall costs of worldwide applications and give users a better experience in terms of velocity and other important aspects of user experience, when a vital Web application is being used on a

daily basis. One of the more significant proposes to solve these known problems was the creation of the Single Page Application (SPA) concept, which means what the name states: an entire Web application would be developed using a single page, and all the subsequent requests from the user would not need to be addressed by the server in terms of processing the entire new page, but only the piece of information required to render in the Web browser.

Usually, any Web application has a master template that contains all the content shared across multiple application pages, such as header, footer, menu, authentication information, and images and other media associated with the pages. During the user navigation through pages, typically, only the central part of the page content is mutable, and the rest of the page remains the same, as demonstrated in *Figure 14.2*:

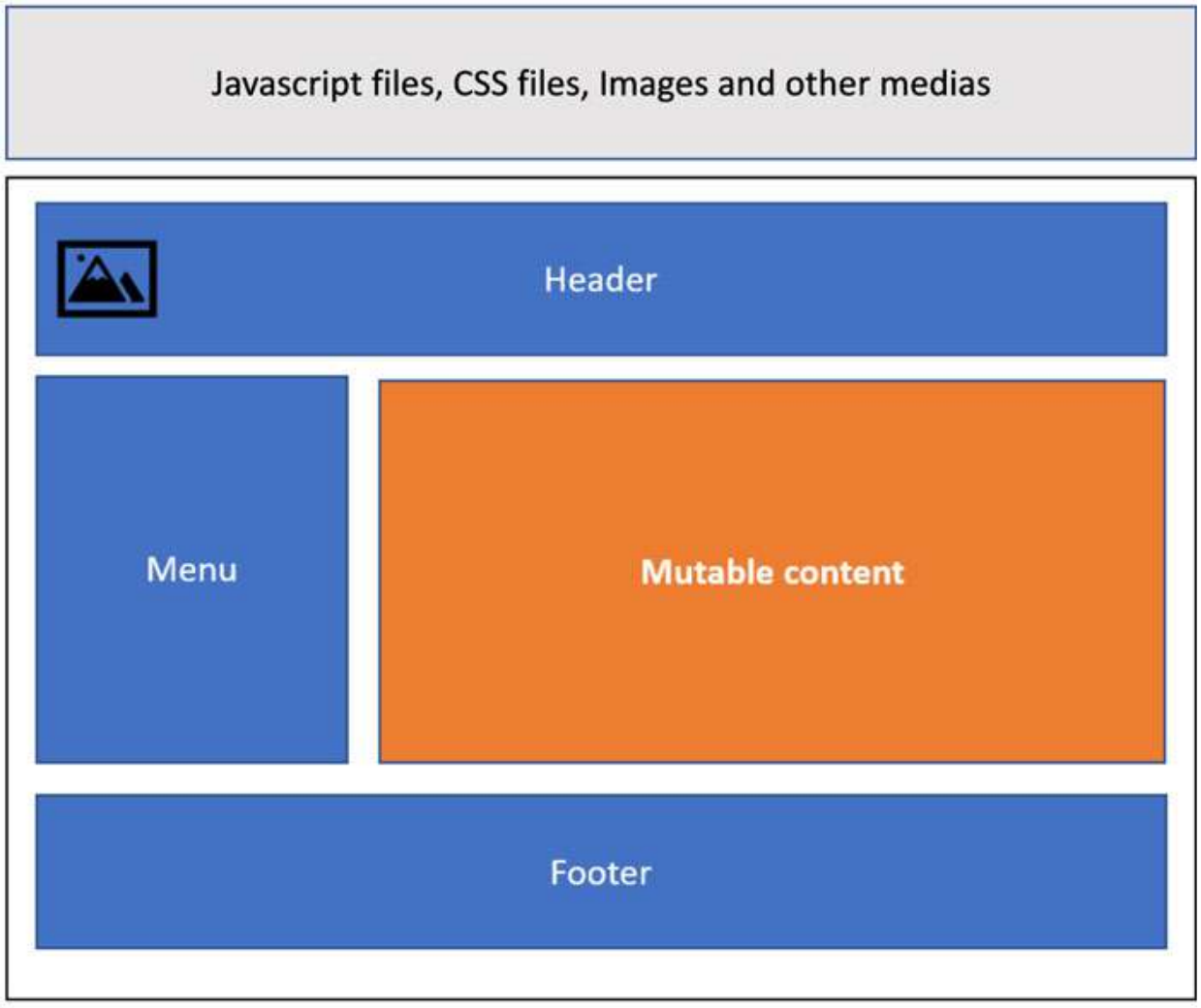


Figure 14.2: *Web page structure*

In many cases, the mutable content part of the Web page represents only a small portion of the entire content displayed to the user. Therefore, in an ideal scenario, when the user navigates across pages, the Web server would return only the part of the content that effectively needs to be rendered in the browser, as the user already has the rest of the content rendered: header, footer, menu, logo, including JavaScript and CSS files. In this context, the Web server processes the request but returns only the necessary data, not the entire HTML with associated content, with all the logical parts to render the HTML constructed on the client side using JavaScript. The most common pattern used for responses between client and server in standard Single Page Applications is the JSON format, which represents an object that can be parsed by JavaScript, C#, and any other modern programming language. If we compare this new Model with the architecture represented in Figure 14.1, the request and response patterns will look like as shown in *Figure 14.3*:

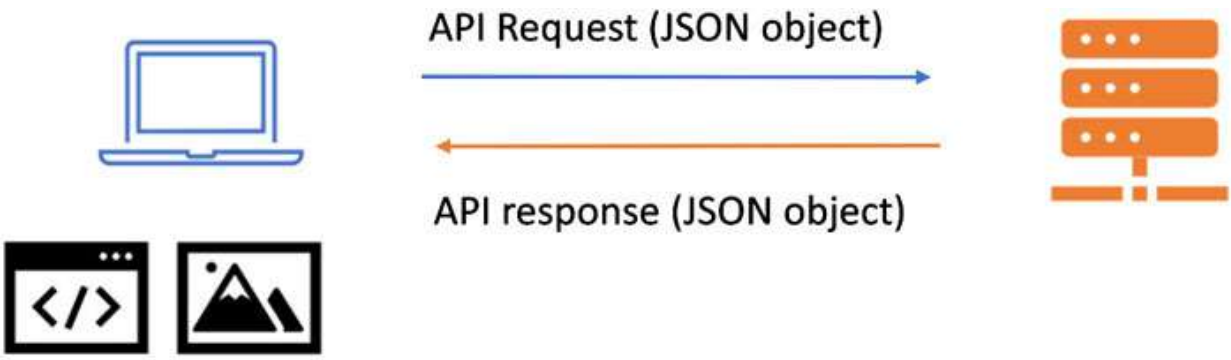


Figure 14.3: *Web response based on JSON*

In the scenario shown in *Figure 14.3*, the Web server returns only a JSON object with the necessary data. The entire logic to render the frontend content is being handled in the front end itself using JavaScript. In cases where the content has a significant size, this approach represents a considerable difference in performance compared to the traditional Web request primarily presented at the beginning of the Web. An entire page was returned from the server on each request.

In this new Model used by Single Page Applications, a JSON object has only a few Kbytes in size, and the cost to build the HTML is done in the browser itself. Therefore, the performance is significantly better. Only the JSON object is transferred via network and Internet from the server to the client, giving users a better experience navigating a website.

Over the last 10 years, numerous Web frameworks based on JavaScript have been built to apply the Single Page Application concepts and give the developers a vast number of ready components and features to increase the performance of the Web applications, such as routes, asynchronous server requests, state management, virtual DOM management, and much more. The most prominent JavaScript frameworks are Angular, React, and Vue JS. They became hugely popular among front-end developers worldwide and were used by millions of professionals. Big companies like Facebook, which created the React framework, keep some of these frameworks. Like Angular and Vue JS, technical communities maintain other frameworks as open-source projects.

All these Web frameworks have a lot of similarities, such as the ability to create components, manage routes, API calls handler, state management, Virtual DOM manipulation, the possibility of using TypeScript, and much more. But the principal characteristic of all of them is that they have JavaScript as the primary programming language, which helped increase the popularity of this language on such a scale that we can't imagine Web development without anything of JavaScript nowadays. All these frameworks reached a high level of maturity and became the foundation for implementing Single Page Applications worldwide.

Since the last versions of the .NET Core platform, Microsoft introduced Blazor, an alternative for creating Single Page Applications, but using C# and WebAssembly as an alternative to Web frameworks based on JavaScript. This new project type is part of the templates available among other Asp.Net Core projects, such as Razor Pages and Asp.Net MVC. Throughout this chapter, you will have opportunities to understand how you can use C# to build Single Page Applications and run C# language directly on the browser, despite it being traditionally a server-side language.

Difference between Blazor Server and Blazor Web Assembly

Blazor represents a considerable innovation within the .NET platform as this new Asp.Net Core project allows us to take advantage of the most modern features for Web development; at the same time, we can still use legacy features from other known project types, such as Razor syntax, Asp.Net Core configurations, dependency injection, model annotations, similarities in

terms of authentication and many other aspects inherited from Asp.Net Razor and asp.Net MVC projects. Additionally, there is the impressive fact that Blazor allows us to run C# directly in the browser. Therefore, it is not required to use JavaScript for Web development with all the possibilities that Web Assembly gives us to run server-side code directly in the browser, including C# language, but Rust, C, and other languages.

As its own name states, Web Assembly represents client development in the browser, while Assembly represents server-side development. In the second case, a high-level programming language is used, like C#, Java, and others, and they can convert to instructions that the machine can understand at a low level, which is assembly itself. Assembly is not easy to understand, and it is not practical to develop directly using it as the productivity for development would significantly decrease. Because of that, high-level programming languages facilitate the development process and serve as middleware between something a human being can interpret and something a machine can understand.

WebAssembly represents the assembly for Web browsers, allowing us to run server-side code within the browser by converting the programming language commands to something the browser can understand. That is a revolutionary concept in Web development as a huge amount of code can be reused and shared between desktop, mobile, and web applications, with the possibilities of libraries and packages being shared for the development of multi-platform applications in an easier way.

In terms of project templates for Web development within the .NET platform, the use of WebAssembly by Blazor represents one of the most significant innovations introduced in the .NET ecosystem since its creation 20 years ago, as it will allow over the next year to develop a single C# application that will be deployable and compatible with multiple devices and platforms in a single code-base, with the same version of the application being used in Windows desktop, macOS, Android, iOS and much more. For a while, Blazor for Web development is already stable and ready to be used in production environments in both ways: Server and Web Assembly.

When opening Visual Studio 2022 and creating a new project, the IDE shows a list of project templates for Web Development, which includes the two types of Blazor applications, as shown in *Figure 14.4*:

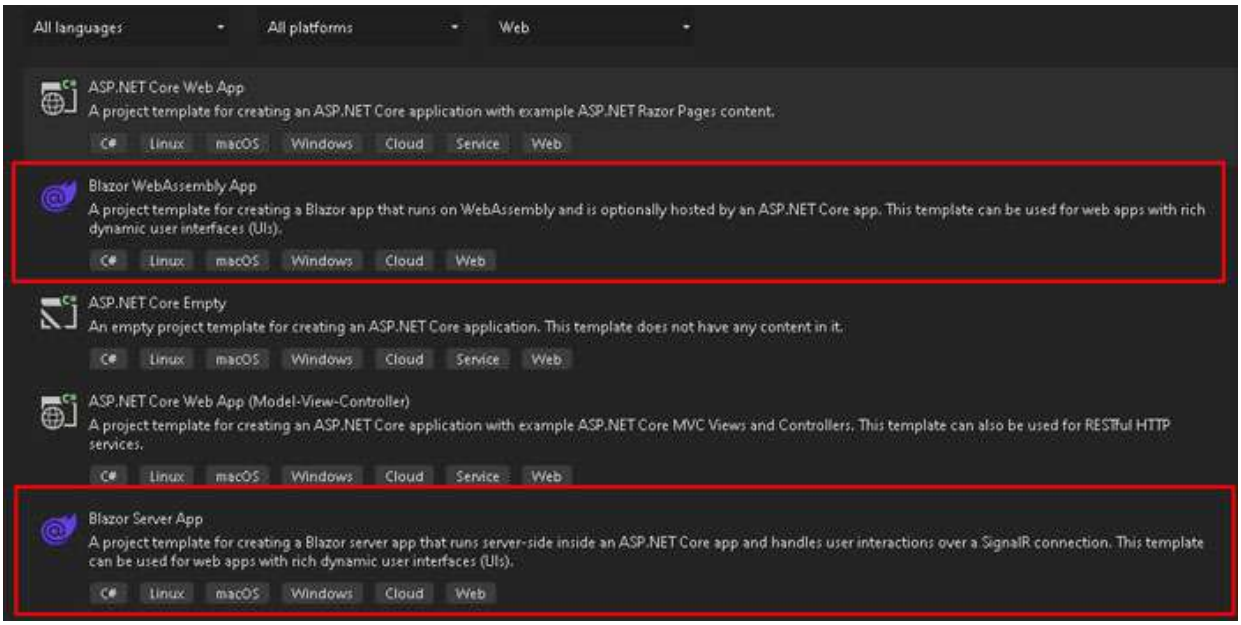


Figure 14.4: *Blazor project types*

The best way to understand these two project types (Server and WebAssembly) is by creating the actual projects and analyzing the general behavior of each application and the project structure. Given that, within Visual Studio, create a new Blazor application by choosing the option "Blazor Server" and give it a name as shown in *Figure 14.5*:

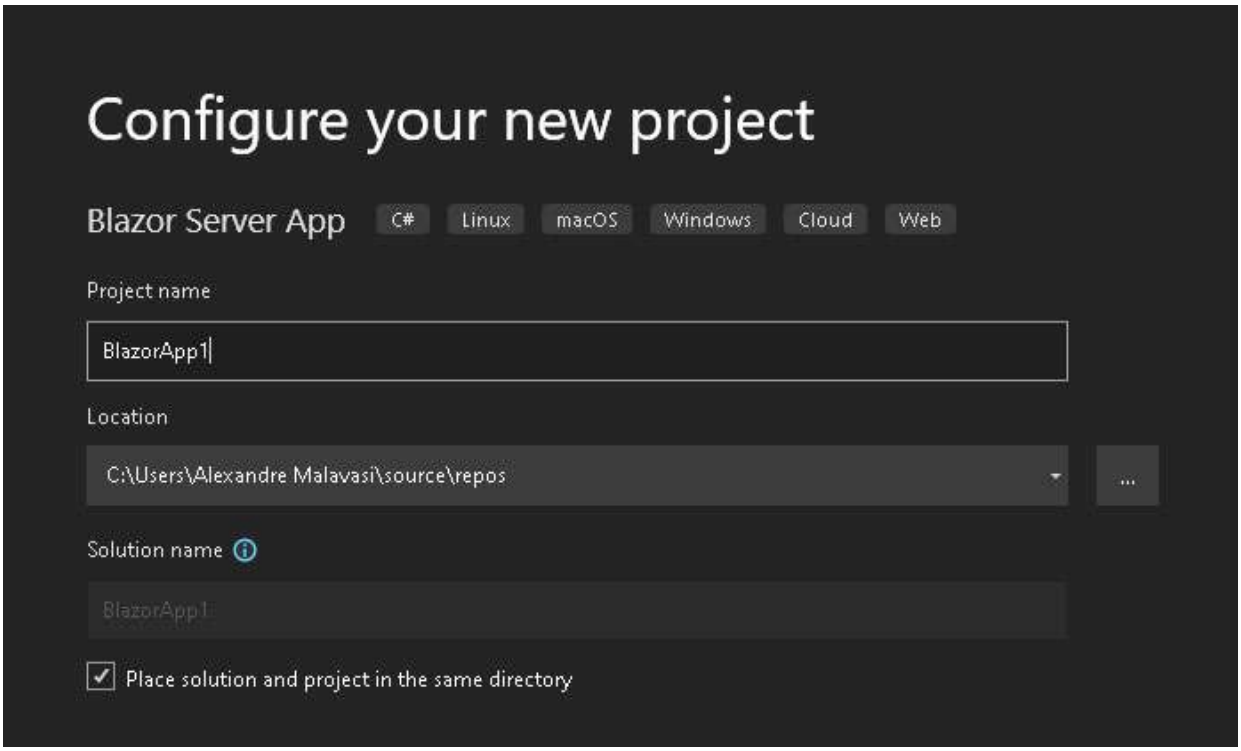


Figure 14.5: *Blazor Server app name*

In the sequence, Visual Studio asks which version of the .NET platform should be used for the project, authentication type, HTTPS, and Docker configurations. In the context of the example in this chapter, the .NET 6.0 version is being used, and the HTTPS and Docker options are disabled, as shown in *Figure 14.6*:

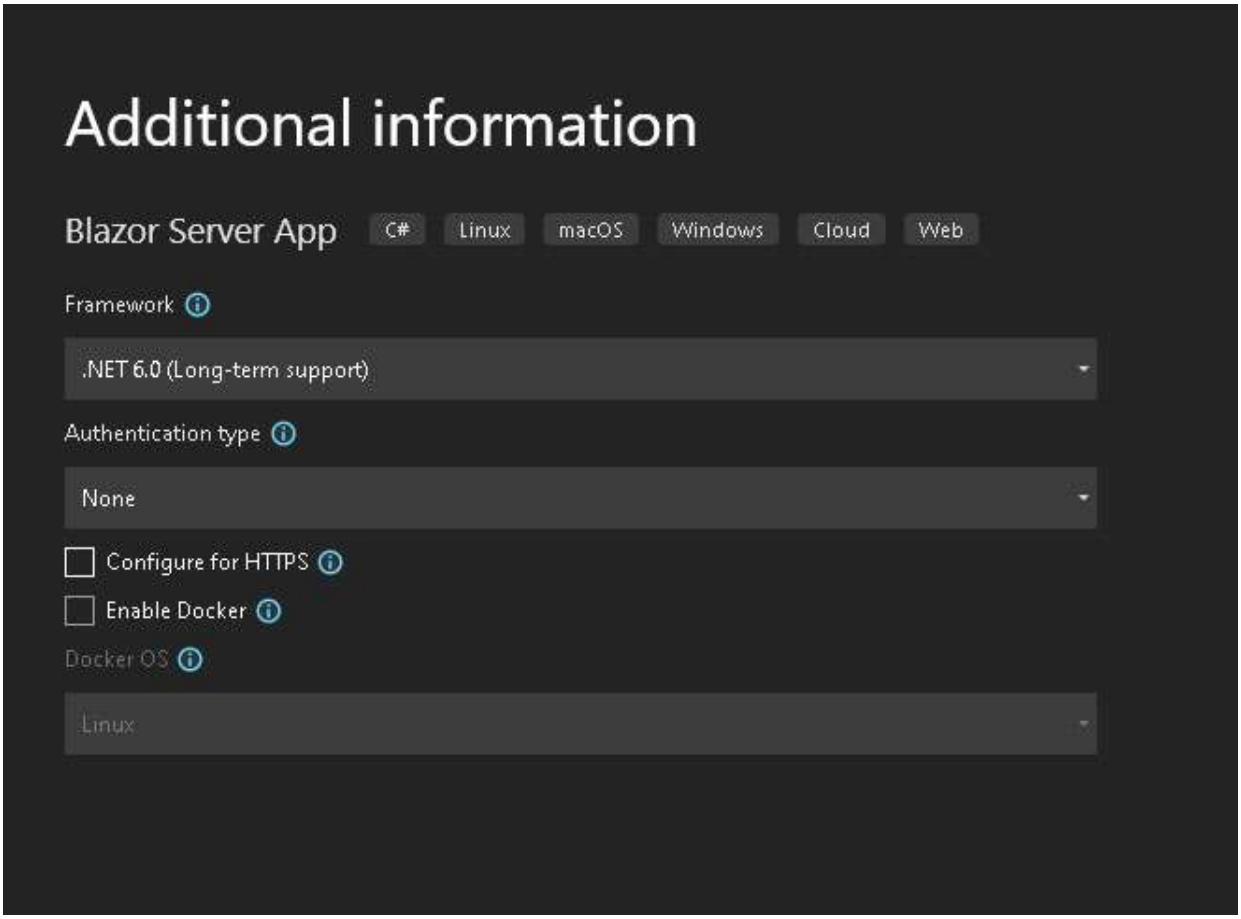


Figure 14.6: *Final configuration for the Blazor Server application*

Once the creation is confirmed in Visual Studio, the project is created in the selected folder with all the files presented on the default project template for Blazor Server, and the file structure should look like *Figure 14.7*:

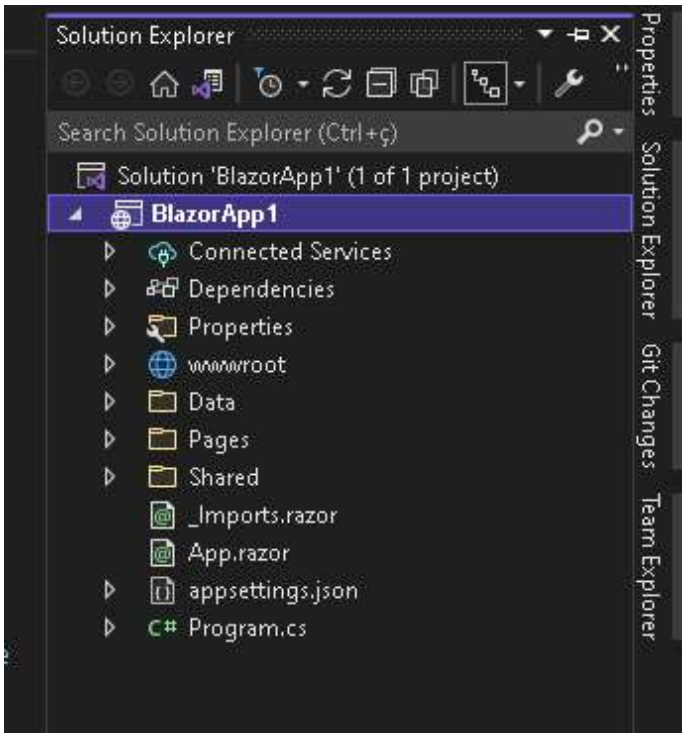


Figure 14.7: *File structure for Blazor Server applications*

The default template for this type of project brings sample pages and components, with the concept of Razor components, similar Asp.Net Core Razor project. Within Blazor, the route configuration for a page is not done within a Controller like for Asp.Net Core MVC projects, but it can be placed inside the component itself. Therefore, if the component has the underlying route directive, it is considered a page and not a component. Blazor Server also uses the known concept of the Layout page used in Asp.Net Razor and MVC projects, allowing us to share common structures for multiple pages. Considering the similarities between Blazor Server and other legacy Asp.Net Core projects, it is possible to speed up the learning process and adaptation of the other projects to Blazor, as it uses Razor syntax within the components and pages, and it is based on C# language as well.

If you run the project on Visual Studio, the Web application will start up with three pages: Home, Counter, and Fetch Data, as shown in *Figure 14.8*:

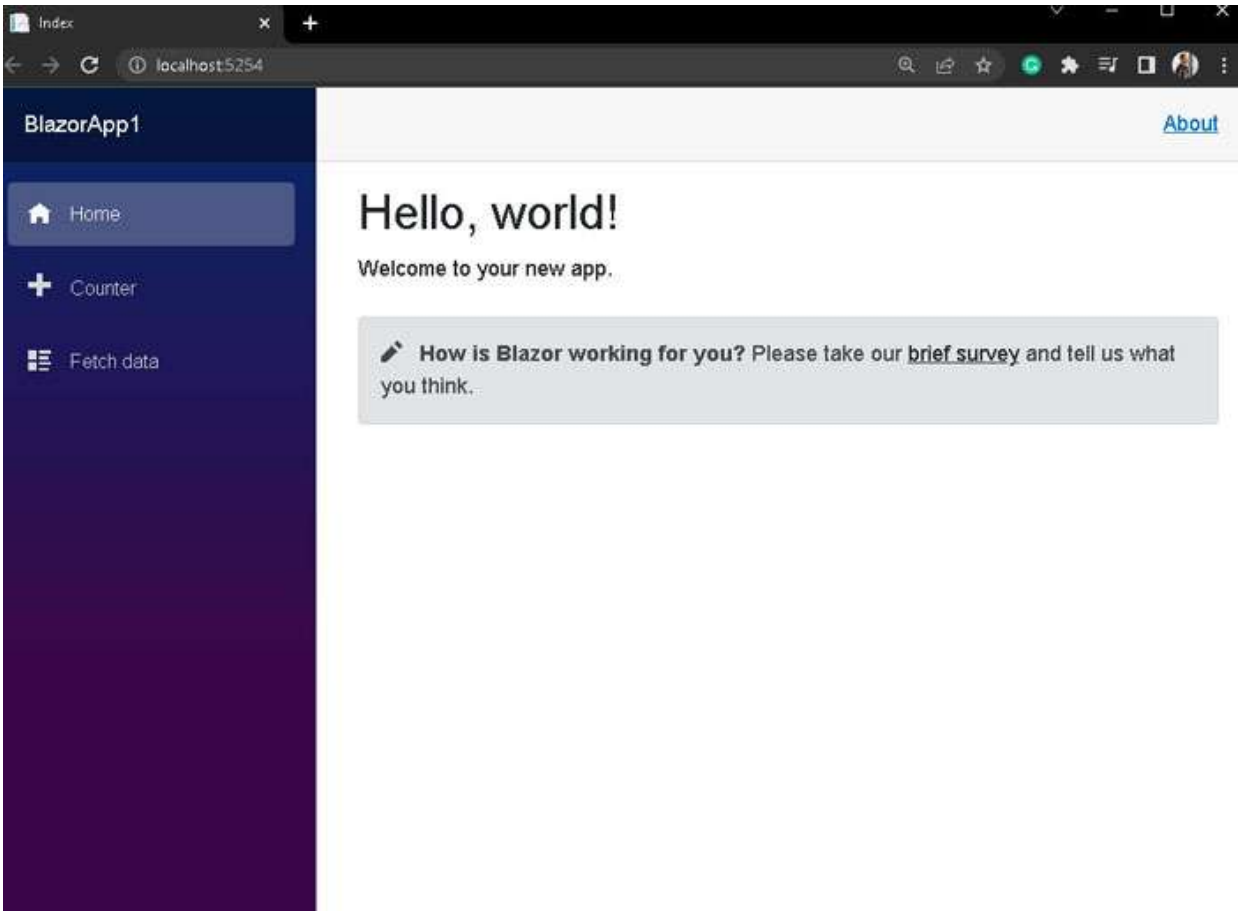


Figure 14.8: *Default project template for Blazor Server*

The entire implementation using C# within this type of project runs in the server, with the components in the browser being updated using SignalR via WebSockets. SignalR is a technology developed by Microsoft that allows real-time communications between the client and server, keeping the communication constantly open throughout the WebSocket protocol. That is different from a standard HTTP request lifecycle when the client and server do not keep the channels alive during the communication. The established connection ends once the server completes the response when a certain request is made. Therefore, the User Interface updates in the Blazor Server project happen through communication between the client and server via SignalR, as demonstrated in *Figure 14.9*:

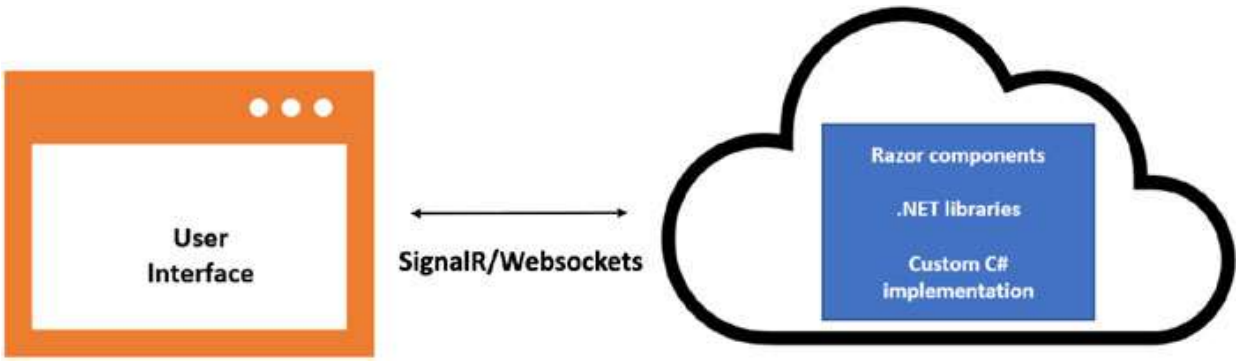


Figure 14.9: *SignalR communication*

In this context, when a user performs any action on the screen that would require interface changes as a response from the backend, the request is processed by the underlying Blazor component in the backend, and the response is synced back to the browser via SignalR using the WebSocket protocol. Suppose you navigate to the Counter page with the application running and inspect the Console in the browser. In that case, a Websocket message is displayed with the details of the constant connection that is established between the client and the server, as shown in *Figure 14.10*:

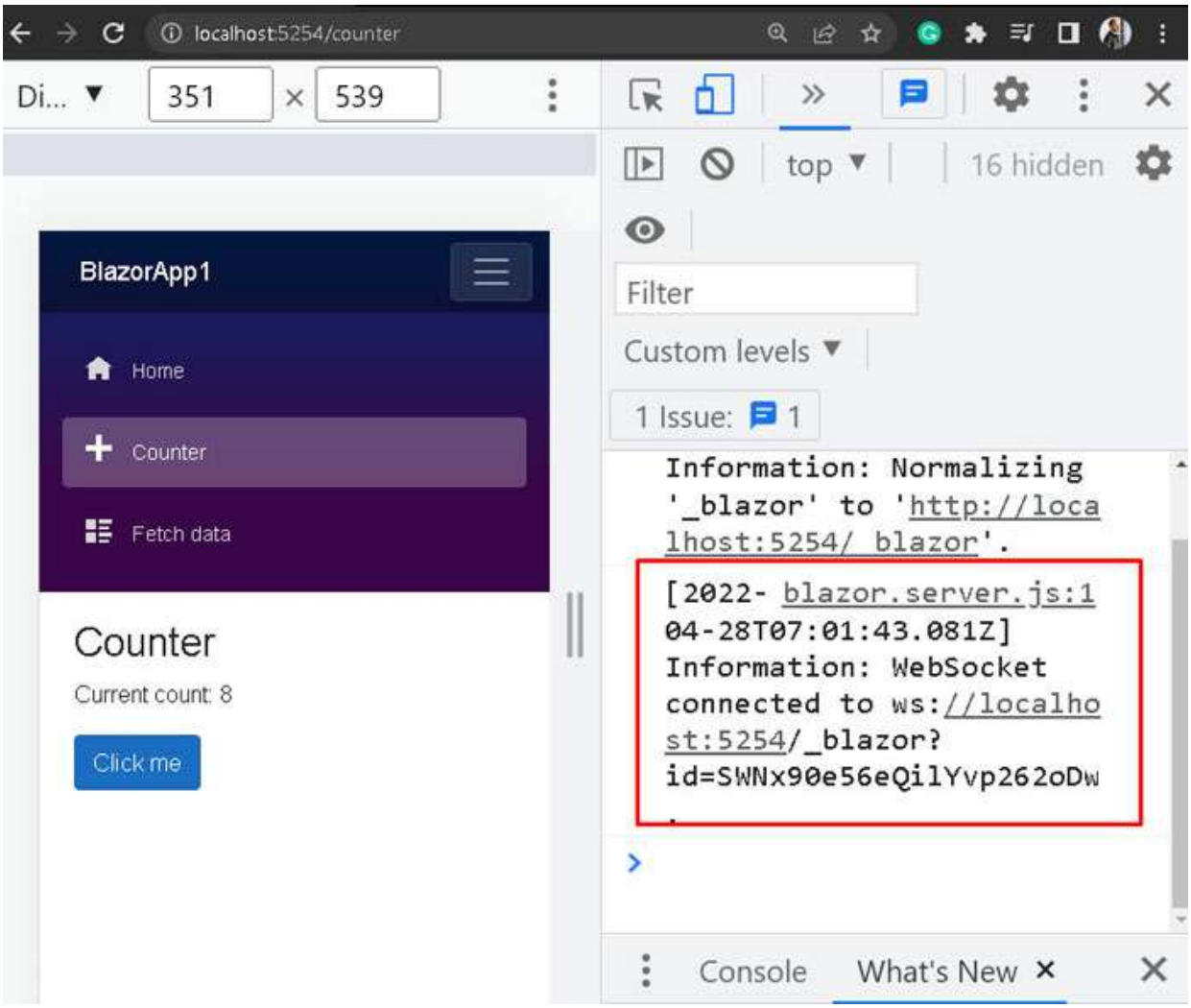


Figure 14.10: *Browser console messages for SignalR*

When using Blazor server applications, each browser tab open represents a new connection opened in terms of SignalR, which means that it represents the use of extra resources from the server. In terms of positive and negative points of Blazor Server applications, if compared to other traditional Asp.Net Core applications such as Asp.Net MVC and Asp.Net Razor, the fact that WebSocket connection is a high-performance protocol is one of the main advantages. However, servers usually have limitations regarding how many simultaneous WebSocket connections can be kept open, which may vary according to the hosting provider. In most cases, the limitation is around 3000 connections for applications hosted in **Internet Information Services** (**IIS**). There is more flexibility if cloud resources such as Azure SignalR are being used to scale and handle a more intense workload. Therefore, the connection limitation factor must be considered when a

Blazor Server app is being developed to determine if the server configuration supports the demand for needed connections. In other words, the use of SignalR by Blazor Server apps represents both the strength and weakness of this type of project, depending on the scenario applied in production environments for real systems.

Once you can understand the main characteristics of Blazor Server applications, it is suitable to understand the fundamental differences compared to Blazor WebAssembly projects. The best way to see those differences is to create a new Blazor application within Visual Studio. Therefore, in the project creation dialog, choose the option Blazor WeAssembly App, as shown in *Figure 14.11*:

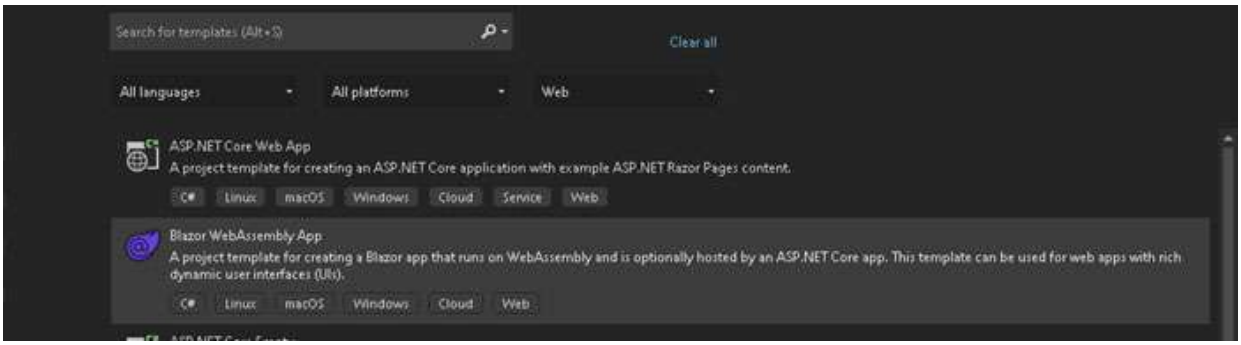


Figure 14.11: *Blazor WebAssembly App*

After giving the project a name, uncheck the “Asp.Net Core hosted” option and choose the most suitable .NET version on the Additional Information dialog. For this chapter, the .NET 6.0 version is being used, as shown in *Figure 14.12*:

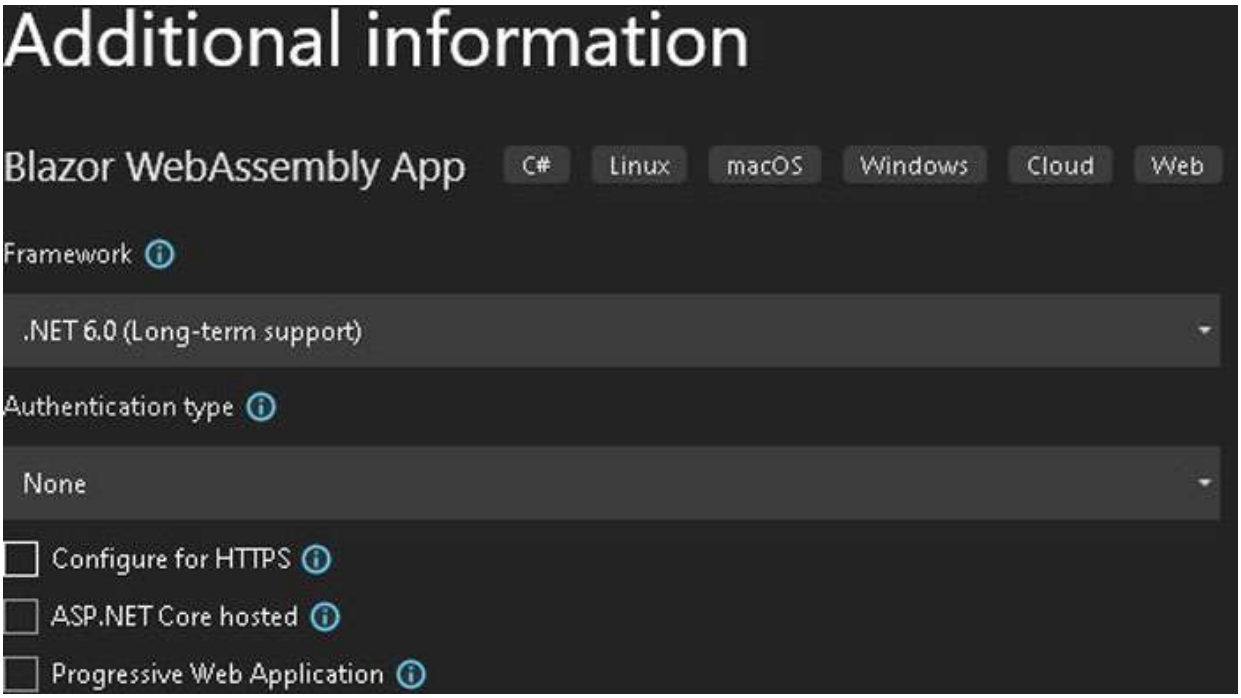


Figure 14.12: *Additional Information for Blazor WebAssembly App*

The folders and file structure within the Solution for the Blazor WebAssembly app looks familiar to the Blazor Server version. However, the behavior of this type of application is entirely different. The first change is the presence of the App Razor component, as highlighted in *Figure 14.13*:

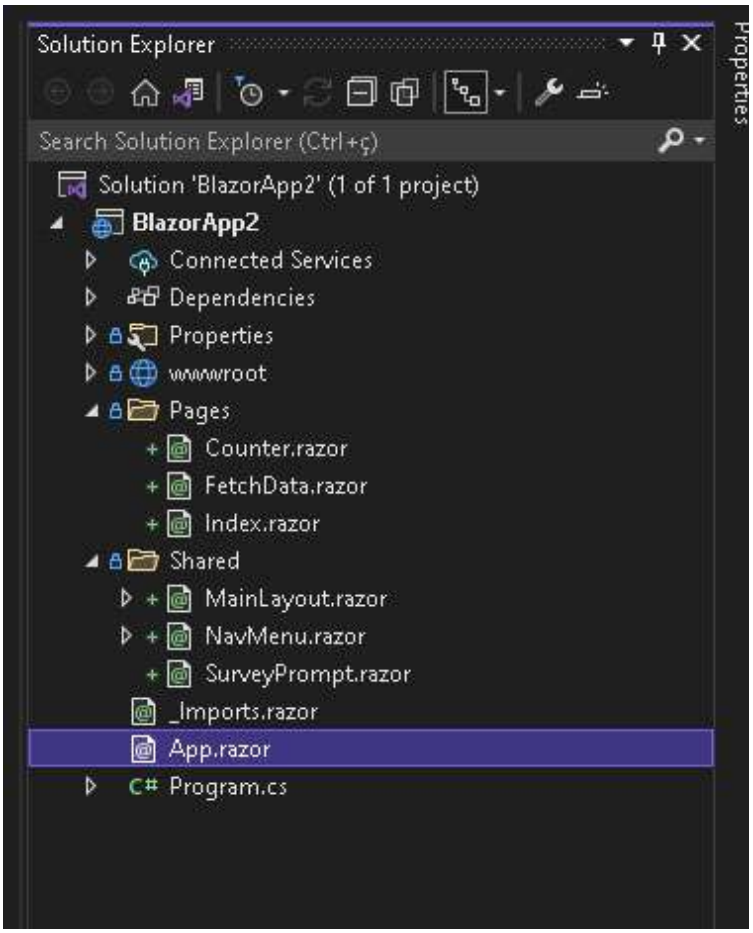


Figure 14.13: *App Razor component*

This file contains the global configuration for all the child Razor components rendered in the Blazor application responsible for `determining` the main layout component that will be the component base for others in terms of design and user interface. Additionally, this main component is the right place to specify authentication aspects demonstrated further in this chapter. By opening the App Razor component that exists in the default template for this type of project, it is possible to find as well pre-configured error handling for request failures in line number six of *Figure 14.14*:

App.razor

```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
    <NotFound>
        <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Figure 14.14: *App Razor component details*

The component configuration for Blazor components follows similar patterns found in other frameworks, such as Angular, React, and Vue JS. These frameworks are based on Web components and allow the relationship between parent and child components as a tree of nested components. It is possible to pass data across multiple instances and levels of the Blazor application, even in more complex scenarios where child components need to update parent components.

After running the application using Visual Studio, at first, there is the impression that this application is identical to the Blazor Serve application demonstrated in this section. However, if you open the Inspection option in your browser, open the Source tab, and refresh the page, you will realize that the application downloaded the blazor.webassembly.js file for the application, as shown in *Figure 14.15*:

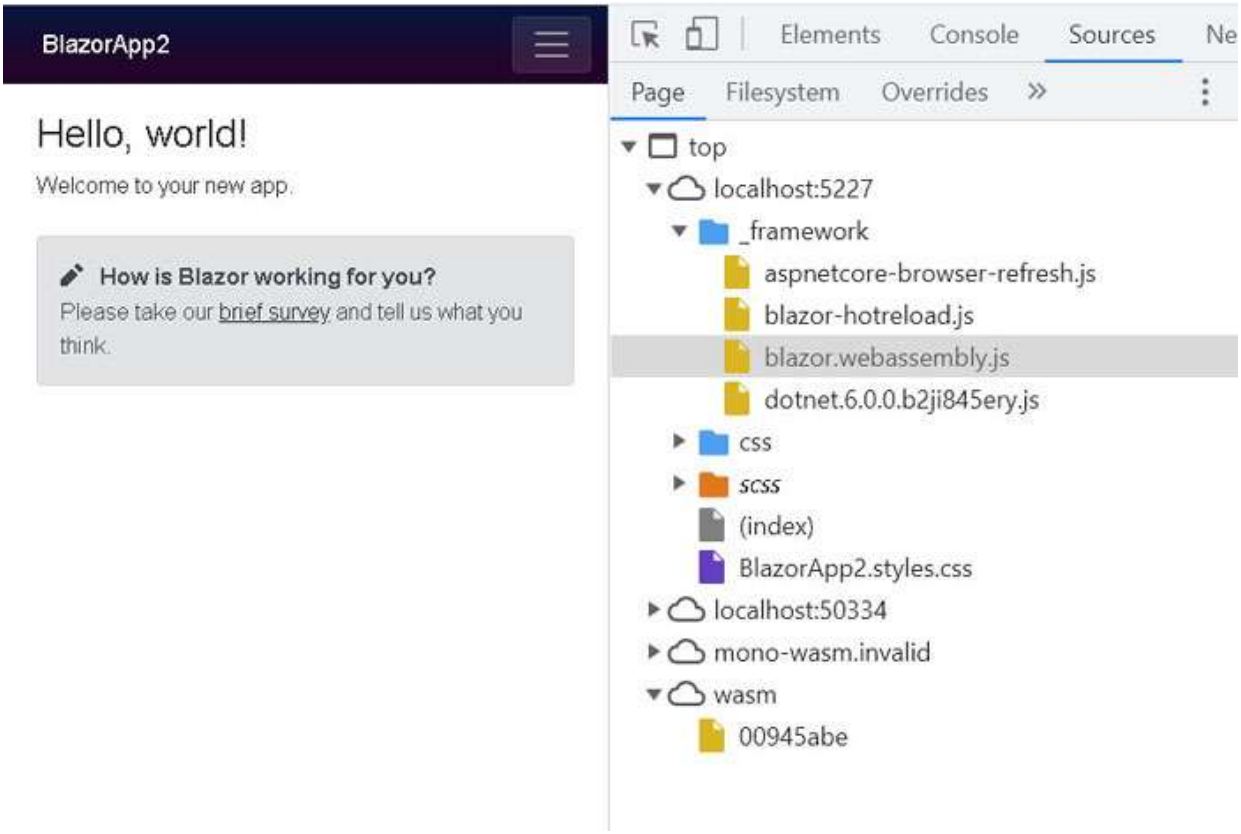


Figure 14.15: *Blazor WebAssembly JS file*

This file contains instructions to allow interoperability between JavaScript and WebAssembly and has all the necessary routines for the application to behave as a Single Page Application. Additionally, the browser downloads a wasm file, which contains all the instructions required for the application to run, even the components are using Razor and C# and not JavaScript per se, as shown in *Figure 14.16*:

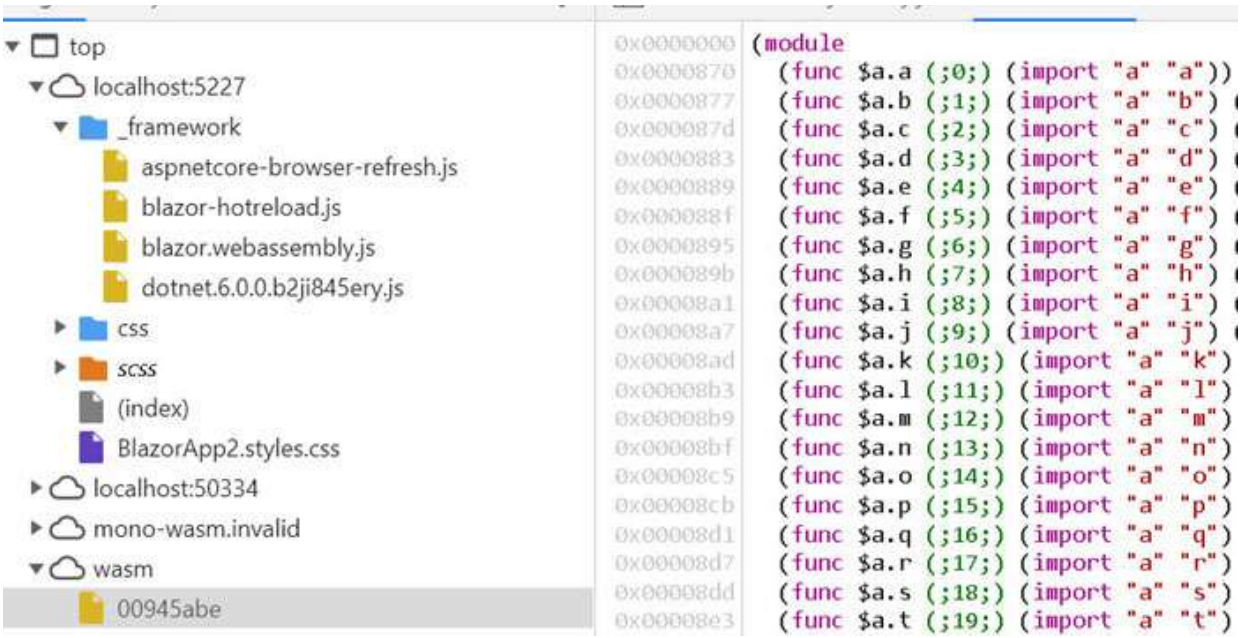


Figure 14.16: *Wasm file*

Wasm is an acronym for WebAssembly. It is a type of file that all modern Web browsers can interpret. The support for WebAssembly is a convention defined by the companies and entities that take part in the W3C organization, which drives the patterns and standards used for the Web in general. As shown in *Figure 14.16* previously, developers cannot read the WebAssembly code. It contains low-level instructions that the browser can interpret and is high-performance, even though it is possible to develop directly in WebAssembly without a middleware framework like Blazor. However, this task would represent a huge challenge and be costly for any project.

One of the advantages of using Blazor WebAssembly is the fact that developers can take advantage of all the power present on WebAssembly, without having to know how to write this type of code as it is possible to still develop in C#, and Blazor will take care of generating the wasm files automatically during the build process.

To evaluate the main differences, advantages, and disadvantages of each of the Blazor apps (Server versus WebAsembly), it is essential to highlight that all the factors present in this chapter may vary based on the context the application is being developed, including functional and non-functional requirements, number of users or workload, infrastructure where the application is deployed, among other essential points.

Given that the Blazor Server has positive characteristics, such as the following:

- The entire process run by the C# language happens in the server; therefore, there is no need for the browser to download any sensitive logic that is encapsulated in the server-side code, representing a good point in terms of security.
- As the app runs on the server, it is easier to access other backend services, such as databases and files, as they can be accessed via an internal network in the server without exposure to an external API.
- The structure of the project is reasonably similar to Asp.Net Razor pages. Therefore, migration to Blazor Server for legacy projects would not represent a huge challenge in most cases. Additionally, the learning curve is significant for development teams with previous experience with other Asp.Net Core projects.
- In cases where a relevant number of potential users have a non-updated Web browser version, it would not have any compatibility issues as WebAssembly is not being used.

In terms of negative points in specific scenarios, Blazor Server contains the following aspects:

- The fact that Blazor Server uses SignalR to sync UI updates from the server can represent a challenge in terms of scalability. Each page by a user keeps the WebSocket connection open with the server, which could be costly to maintain.
- UI updates may have instability as it depends on the latency of the connection between the client and the server via SignalR.

As we walked through the pros and cons of Blazor Server, it is vital to highlight that all these factors and aspects may vary depending on the application's development scenario, such as functional and non-functional requirements.

The Blazor WebAssembly also has advantages and disadvantages compared to Blazor Server or other Asp.Net Core projects. In summary, the pros and cons are quite the opposite of Blazor Server, but with the following highlights in terms of positive points:

- It is scalable as a huge amount of the process runs in the Web browser, relieving the server of resources.
- As this type of project is a Single Page Application, it can be hosted on a less expensive server once the Web browser downloads all the necessary files and runs them on the client side.
- Because the application runs on the client side, there are no latency costs.
- WebAssembly per se is a high-performance technology.
- It can run offline using the Progressive Web Apps mode.

In terms of negative points, Blazor WebAssembly may present the following factors:

- The first download of the application may be more costly than other Single Page Applications like Angular, React, and Vue JS.
- All modern browsers fully support WebAssembly. However, many users may still use a non-updated browser version, therefore incompatible with this technology.
- As it does not run on the server, all the communication backend services must be done through APIs. Sensitive data cannot be part of the application code, as reverse engineering can take this information.

Razor components and data binding

One of the most iconic characteristics of any Single Page Application framework is the easy way to create reusable Web components. Blazor does not get behind on that point as it allows us the creation UI components using Razor.

Usually, a Web component contains HTML, CSS, and the logical code to handle the component renders, event handler, state management, and component lifecycle. The logical code is expressed in C# and Razor within a Blazor component, and interoperability with JavaScript is possible, allowing developers to integrate Blazor WebAssembly with a broader range of libraries.

To create your first Blazor component, create a new Blazor WebAssembly project within Visual Studio and, in the Solution Explorer, right click on the

Shared folder, and choose the option Add and Razor Component, as shown in *Figure 14.17*:

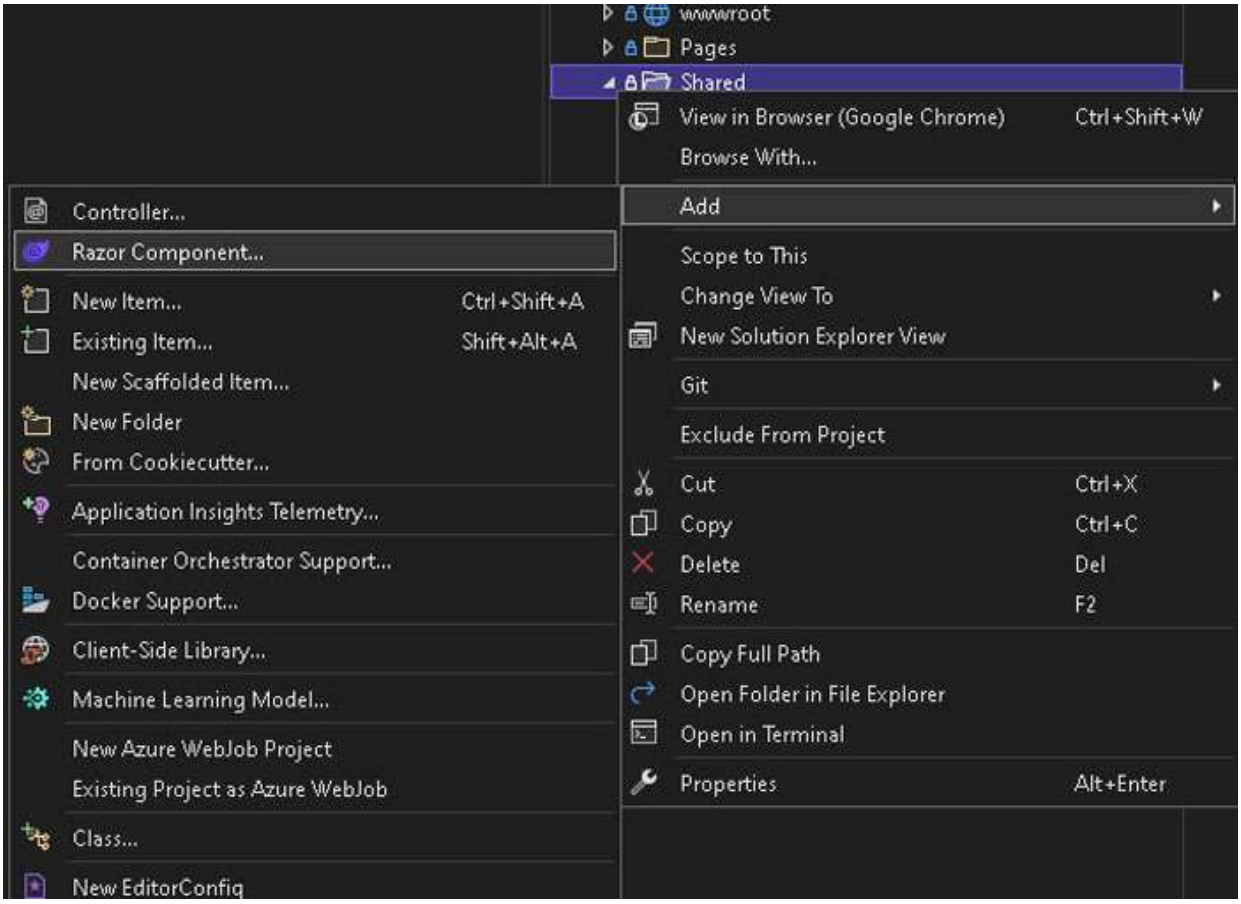


Figure 14.17: *Razor component creation*

For example purposes, the component's name in this section is “MyFirstComponent.” After creating, the component contains a code section where all the logical implementations using C# can be placed. Additionally, in the rest of the components, it is possible to specify route configuration, parameters, HTML, CSS, and Razor statements to handle the component state.

To verify that your component is working as expected, go to the **`Index.razor`** file under the Pages folder, and create an instance of the new component, as shown in *Figure 14.18*:

Figure 14.18: *Razor component reference*

As shown on line number five in *Figure 14.18*, the component is referenced following the exact same name as the name of the Razor file created for the same component. The syntax is quite similar to the one used by other Singla Page Application frameworks based on JavaScript, allowing an excellent adaptation for developers who have experience with Web frameworks other than Blazor.

Additionally, Blazor allows us to work with one-way and two-way data binding. In the first case, the UI is rendered respecting the value in the underlying Model, which can be a property of the component passed by a parent component or an internal variable within the component. Using our "**`MyFirstComponent`**" creates a local variable giving any assignment and specifies the variable in the HTML using Razor, as shown in *Figure 14.19*:

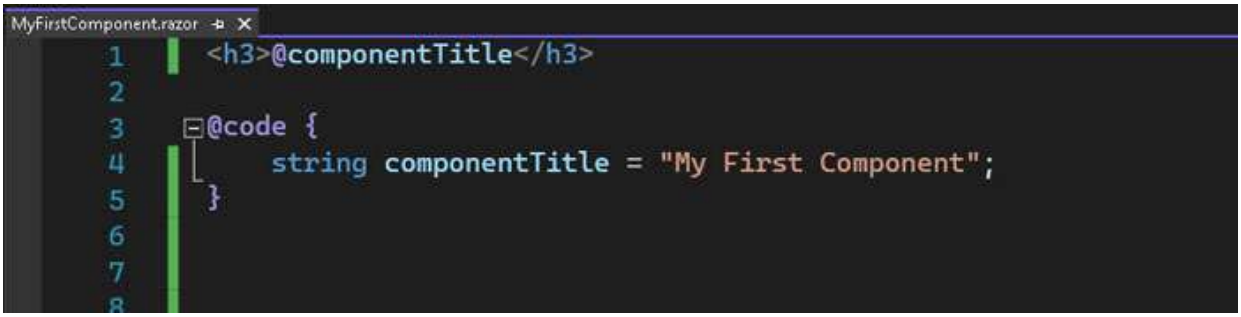


Figure 14.19: *One-way data binding*

In this case, once the component loads, the component renders the content presented in the "**`componentTitle`**" variable within the HTML part of the component. Therefore, the Model has a direct effect on the UI. If you run the application, you will get the result as presented in *Figure 14.20*:

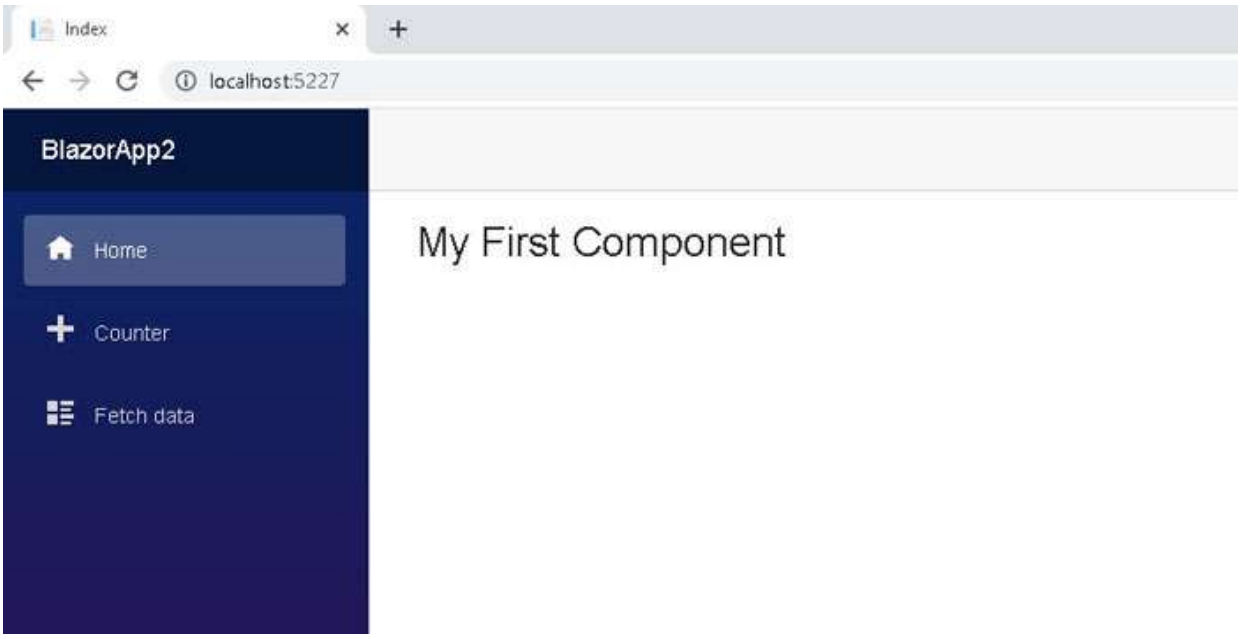


Figure 14.20: *Component render with one-way data binding*

On the other hand, there is the two-way data binding mode for Razor components, which means that the Model affects the UI, but the UI can also affect the Model state. This is particularly useful when the application needs to have Forms or any other type of construction that allows the user to input data of some kind.

To test and verify how the two-way data binding works within the same component, create a text field specifying the "**`componentTitle`**" variable as the bind associated with the component, as shown in *Figure 14.21*:

```
Counter.razor   MyFirstComponent.razor
1   <h3>@componentTitle</h3>
2
3   <p>
4       Component title:  <input value="@componentTitle"
5               @onchange="@((ChangeEventArgs __e) => componentTitle = __e?.Value?.ToString())" />
6   </p>
7
8   @code {
9       private string componentTitle = "My First Component";
10
11      protected void UpdateValue()
12      {
13          StateHasChanged();
14      }
15  }
```

Figure 14.21: *Two-way data binding*

The input text field has an "**onchange**" event associated, which triggers the UI update once any content is typed into the text field and the cursor changes

the focus to another point on the screen. Like what happened with JavaScript events associated with HTML controls, Blazor contains an extensive list to trigger many events to manipulate the component's state and give the application a rich user experience. As shown in *Figure 14.21* on line nine, the local variable "componentTitle" was changed to private property in this component's context. After running the application and typing any content under the input text, the component title under the tag "**<h3>**" automatically changes, as shown in *Figure 14.22*:

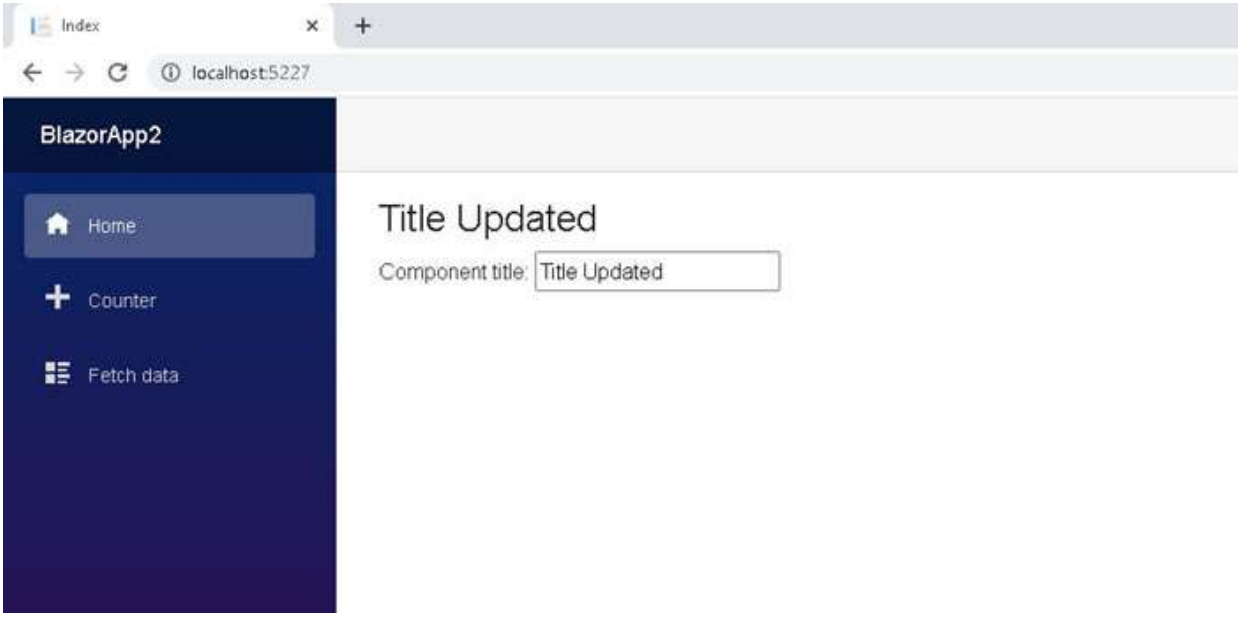


Figure 14.22: *Two-way data binding update*

Additionally, the state of the component can change based on component parameters passed from parent to child components, which is quite related to the one-way and two-way data binding approach, with the only difference that the state is being changed by something external to the component itself, as the content for the component properties is passed by the parent component. To adapt our "**`MyFirstComponent`**" to use component parameters, change the "**`ComponentTitle`**" variable to be an actual property with "**`get/set`**," specify the "**`Parameter`**" annotation, and update the Index.razor page to pass the component title, as shown in *Figure 14.23*:

```razor
<h3>@ComponentTitle</h3>


@code {
    [Parameter]
    public string ComponentTitle { get; set; } =  "My First Component";
}
```

Figure 14.23: *Component parameter*

In this example, the "**`ComponentTitle`**" was changed to a capital letter to follow a more conventional syntax for parameters, and the references to this property were updated on the HTML. Furthermore, the input text was removed to understand the example better. The Index.razor page that has a reference to the "**`ComponentTitle`**" component was also changed to pass the title, as shown in *Figure 14.24*:

```razor
@page "/"

<PageTitle>Index</PageTitle>

<MyFirstComponent ComponentTitle="Component Title via Parameter" ></MyFirstComponent>
```

Figure 14.24: *Parameter on Index.razor*

Component parameters can receive simple information as a string or even complex objects, and it is possible to configure which parameters are mandatory or not. Rerunning the same application after the changes, you will get the following result, as presented in *Figure 14.25*:

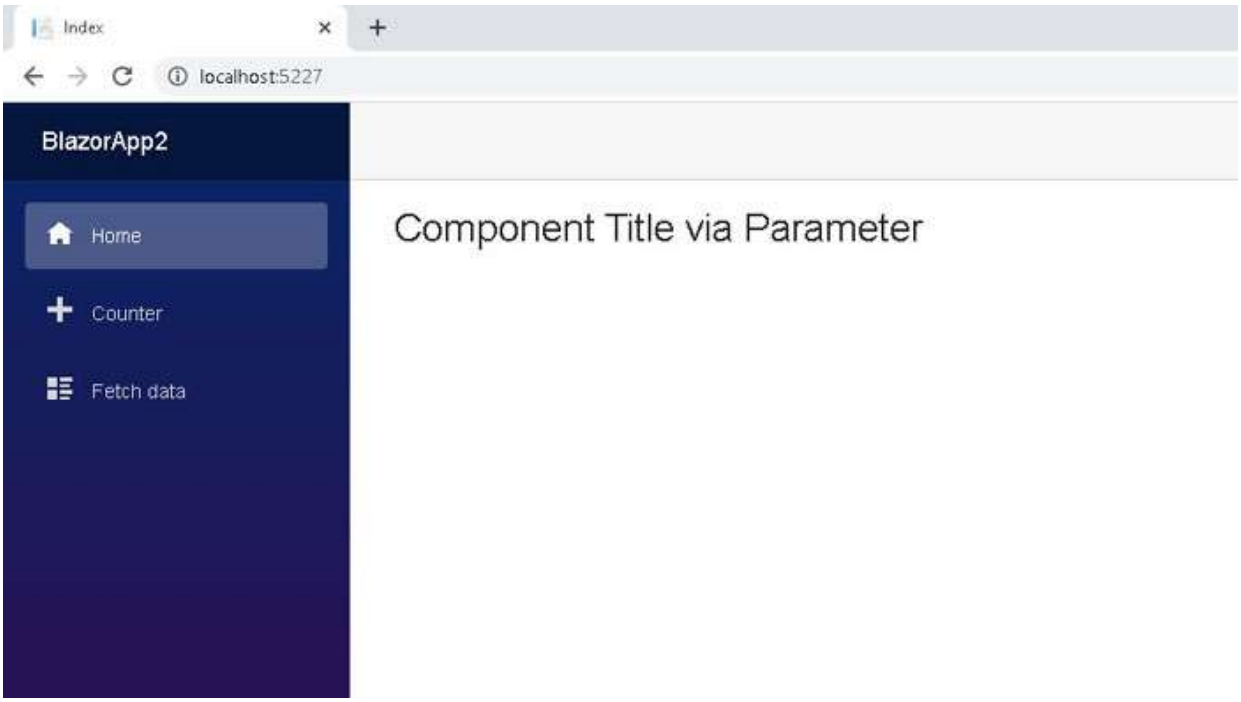


Figure 14.25: *Component title via parameter*

Component parameters are helpful to increase the reusability of components as the state becomes flexible with the data being passed from a parent component to a child component. In more complex cases, it is possible to update a parent component's state from a child, propagating the changes across the tree of components. However, good practices in terms of simplicity are extremely recommended regarding dependency and coupling between components because whatever is done directly affects reusability, testability, and maintainability.

JavaScript Interop

Even though Blazor is based on C# and Razor, it is a fact that frameworks based on JavaScript are much more popular. Therefore, a much wider range of libraries is available, maintained by private companies, or even kept as open-source projects. Given that, it is reasonable to think that enterprise Blazor applications in real scenarios must integrate with JavaScript.

Blazor allows us to inject a service that implements the IJSRuntime interface to establish communication between C# and JavaScript. The implementation presents two main methods that can be used to call JavaScript functions from C#: **InvokeAsync** and **InvokeVoidAsync.** These two methods can be used

to call predefined JavaScript functions and execute any JavaScript code standalone, such as alerts, console logs, and any other JavaScript feature.

To verify how the interoperability works within the previous Blazor WebAssembly project, create under the Shared folder a Razor component called “JavaScriptExample.razor” and specify the code as shown in *Figure 14.26*:

Figure 14.26: *JavaScript Interop example*

As you can see in the first line of the component, an object that implements the IJSRuntime interface is being injected. The JSTestClick method invokes a JavasScript alert passing the message “Blazor Interoperability.” C# language does not have features to trigger alerts and other browser interactions. Instead, it needs to use JavaScript to perform these operations. In the Index.razor page, create a reference to the new component, as shown in *Figure 14.27*:

Figure 14.27: *JavaScriptExample component reference*

After running the application and clicking on the JSTest button, you should be able to see the alert working, as shown in *Figure 14.28*:

Figure 14.28: *Component title via parameter*

The same interoperability can be used to call any custom JavaScript functions the application needs to run. The underlying JavaScript file is referenced under the wwwroot/index.html file, including traditional JavaScript libraries for Maps, Charts, and others. Additionally, within the C# function, it is possible to pass parameters to a JavaScript function using C# objects. It is also possible to receive a response from JavaScript to serialize that with compatible object types in C#.

Conclusion

In this chapter, you had a brief overview of Blazor, the difference between the Server and WebAssembly project types, and you could understand the fundamental concepts of Single Page Applications.

Blazor represents a robust Web framework that allows us to create reusable components using Razor syntax and C# language, and at the same time, it gives us the possibility of running C# directly on the Web browser via WebAssembly, which represents one of the most significant innovations in .NET since its creation 20 years ago.

In the next chapter, you will have the opportunity to learn about desktop, console, and mobile applications. NET.

Points to remember

- There are pros and cons in terms of Blazor project types.

- Blazor Server uses SignalR to facilitate communication between the client and server for UI updates.
- Blazor WebAssembly represents a Single Page Application like Angular, React, and VueJS. However, it is based on C# and not on JavaScript.
- It is possible to establish interoperability between C# and JavaScript in Blazor WebAssembly projects.

Multiple-choice questions

1. **Which alternative does not represent a positive point of Blazor Server?**
    a. No browser compatibility issues
    b. Easy integration with backend services, like databases
    c. The high learning curve for developers with experience with Razor pages
    d. UI updates via SignalR
2. **Which alternative represents a negative point for Blazor WebAssembly projects?**
    a. It runs entirely on the client-side
    b. Scalability
    c. Risk for sensitive configuration data, such as connectionstrings and tokens
    d. None of the above

Answers

1. **d**
2. **c**

Questions

1. Explain the main characteristics of Single Page Applications.

2. Using what you have learned, create a new Blazor WebAssembly project, calling your JavaScript function within a component.
3. In which cases or scenarios the Blazor Server would be suitable?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 15

Desktop, Console, and Mobile Applications

Introduction

One of the main characteristics of the .NET platform since the .NET Core 1.0 version is the fact that the entire platform moved to be multi-platform compatible, allowing developers to create applications that would be able to run on multiple devices, operating systems, and platforms without so much relevant code change, using only C# language. It includes the development of Desktop, Console, and Mobile applications using the .NET platform.

Learning what the .NET platform supports for native C# applications for Desktop and Mobile will allow you to make proper technical decisions for projects that would be more suitable to run, not only for the Web but also for personal computers, cell phones, and other devices as well, giving you a broader vision of the .NET capabilities for enterprise applications development.

This chapter will cover the most relevant project types in .NET that represent an alternative to Web and Cloud development, focused on Desktop, Console, and Mobile development and theoretical concepts.

Structure

In this chapter, we will discuss the following topics:

- Native application development
- Mobile development
- Console applications

Objectives

After studying this unit, you will understand what can be done in .NET and C# in terms of Desktop development and create basic Console Applications using C#. You will learn how to make technical decisions based on the project types available for Desktop and Mobile development. You will also gain essential knowledge on Mobile development within the .NET platform15.1 Desktop applications evolution.

Native application development

Despite the rapid rise of the Internet over the last 20 years, applications that are not web-based are still quite common for enterprise applications. Many developers globally are still doing software development for Desktop applications. Long-running tasks, complex ERPs, and applications that require a more rich interaction with the user's personal computer are the main reasons why Desktop development is still popular and, we can say, extremely necessary in many scenarios.

For that reason, Microsoft, other companies, and open-source project organizations continued to provide relevant updates to the .NET platform and C# to support the most important demands from the market, such as multi-platform development for Linux, Windows, and macOS, and the introduction of new features that would be compatible with the most modern devices.

Web development allows us to create rich applications for sure. Still, they are restricted to the Web browser capabilities and the security concerns established for this kind of application. Desktop development contains many benefits for specific scenarios such as:

- Integration with more complex features, such as Bluetooth, USB, and others
- Better use of the operating system and CPU capabilities in the client
- More stability as it does not necessarily depend on the Internet or networking connection for certain operations
- It is much more mature in terms of software development practices, as Desktop applications have been in the market much longer than web applications

The .NET platform adapted itself over the years to meet the market demands for Desktop applications, starting with **Win32** based application in late 1995, passing through **Windows Forms** technology from 2002, and a little modernization beginning in 2006 with **Windows Presentation Foundation (WPF**) and finally with **Universal Windows Platform (UWP**) since 2016.

Windows Forms

Windows Forms represented a considerable accomplishment within the .NET platform around 2002. It was combined with excellent features within Visual Studio that allowed the developers to drag and drop visual components into the screen, with an incredible amount of ready components in the Toolbox to choose from. The high-quality usability of Visual Studio helped Windows Forms to become hugely popular. Within a couple of years, the market already had millions of applications being developed by small, medium, and large companies, representing huge market participation compared to more traditional frameworks, such as Java and C++.

Despite other options available within the .NET platform, Windows Forms was still supported for updates, including .NET Core, .NET 5, and .NET 6 versions. To create a Windows Forms project, go to Visual Studio and choose the option Windows Form in the Project Creation dialogue, as shown in *Figure 15.1*:

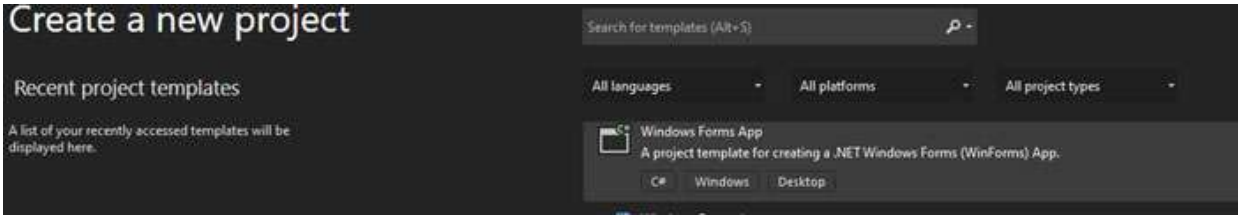


Figure 15.1: *Windows form project creation dialog*

After giving the project a name, choose the .NET version to create your Windows Form project. In the context of this section, this book uses the .NET 6.0 version, as shown in *Figure 15.2*:

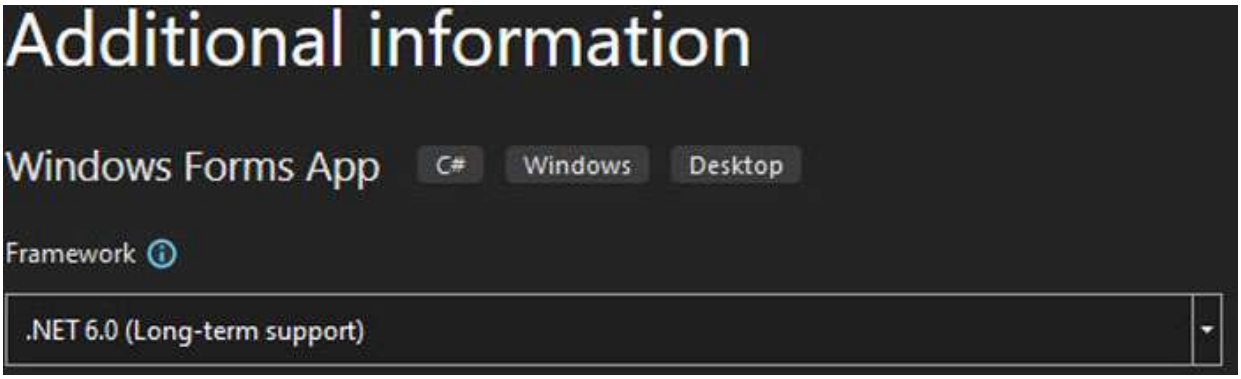


Figure 15.2: *.NET version for Windows Form*

The .NET platform supports creating Windows Forms applications using Vb.NET and not only in C#. As the C# language will have support and new features from .NET 7 and not Vb.NET, C# was chosen as the primary language for all the examples in this book.

If any of the project templates shown in this chapter are not available in your Visual Studio, make sure you have the following workloads installed along with it:

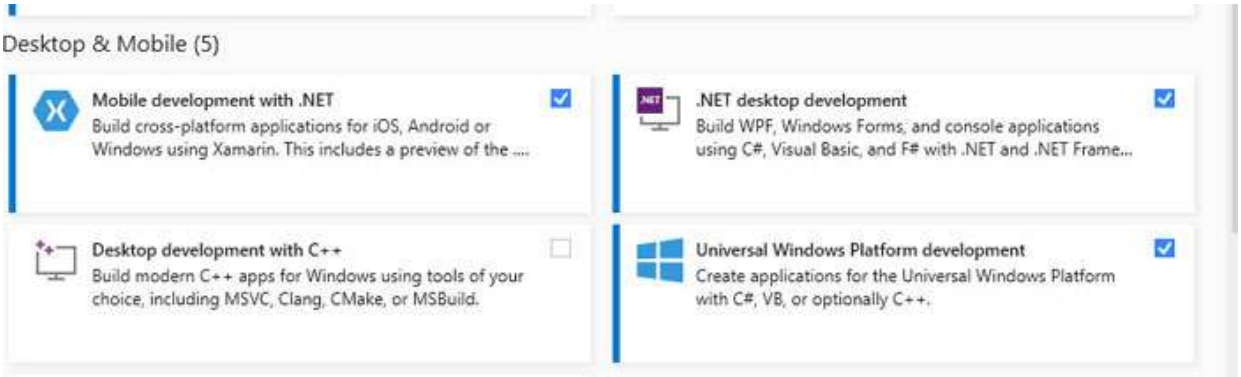


Figure 15.3: *Workloads for Windows Forms*

After confirming the creation of the Windows Forms project, Visual Studio shows a Design mode with a default form where new components can be added from the Toolbox in Visual Studio, as shown in *Figure 15.4*:

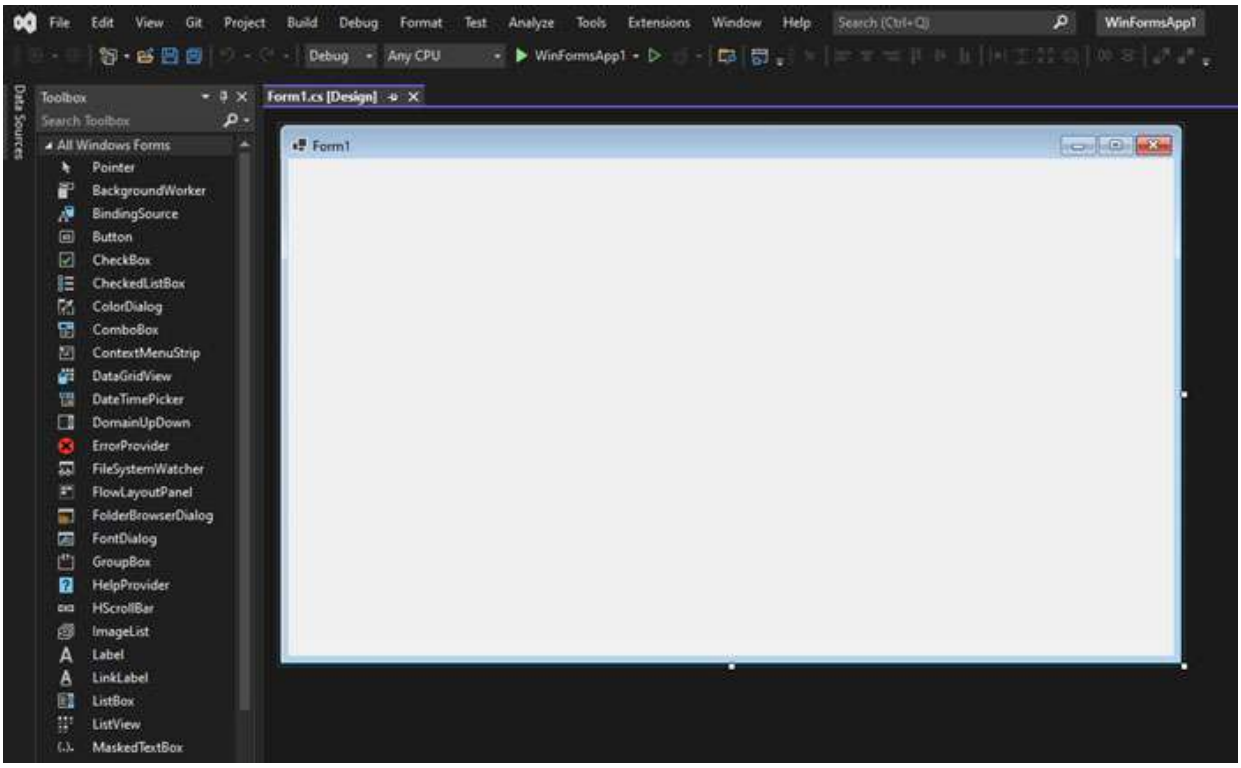


Figure 15.4: *Default Windows Form Design Mode*

Visual Studio contains a huge list of Windows Forms components that speed up the development of this type of application. To use any component, you need to drag and drop the element in the main form, as shown in *Figure 15.5*:

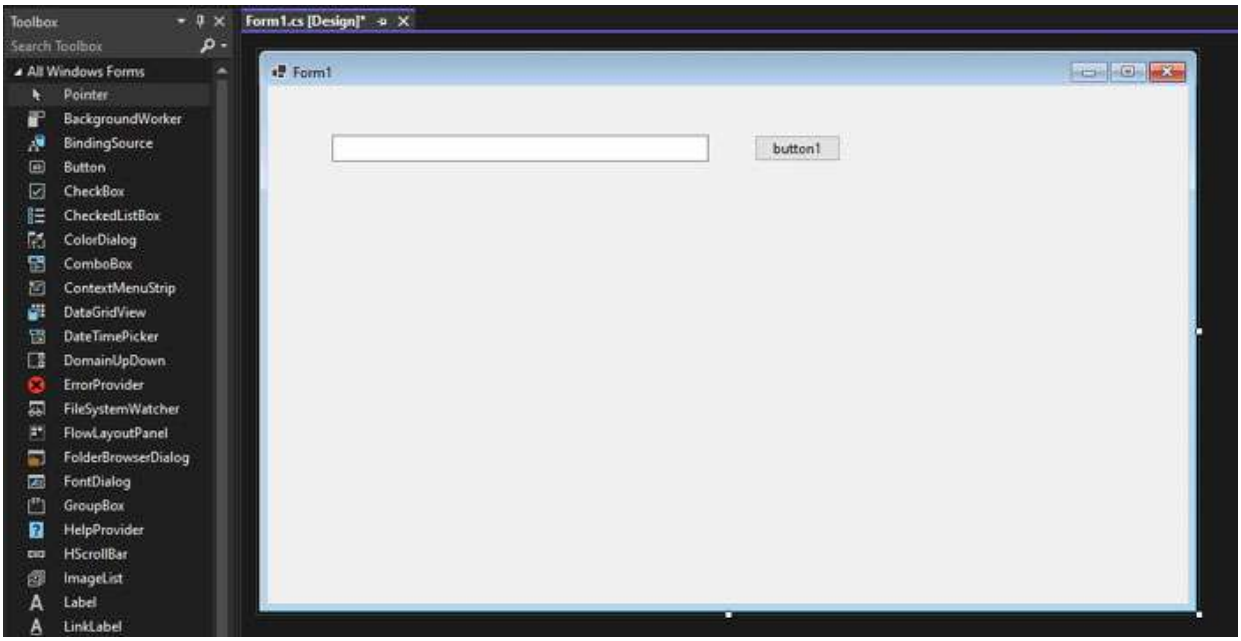


Figure 15.5: *Windows Forms components*

Each component has common properties and specific ones particular to the component type. Select a particular component on the screen, for instance, the button component on the left sidebar on Visual Studio. You should be able to see the underlying properties of the component, as shown in *Figure 15.6:*

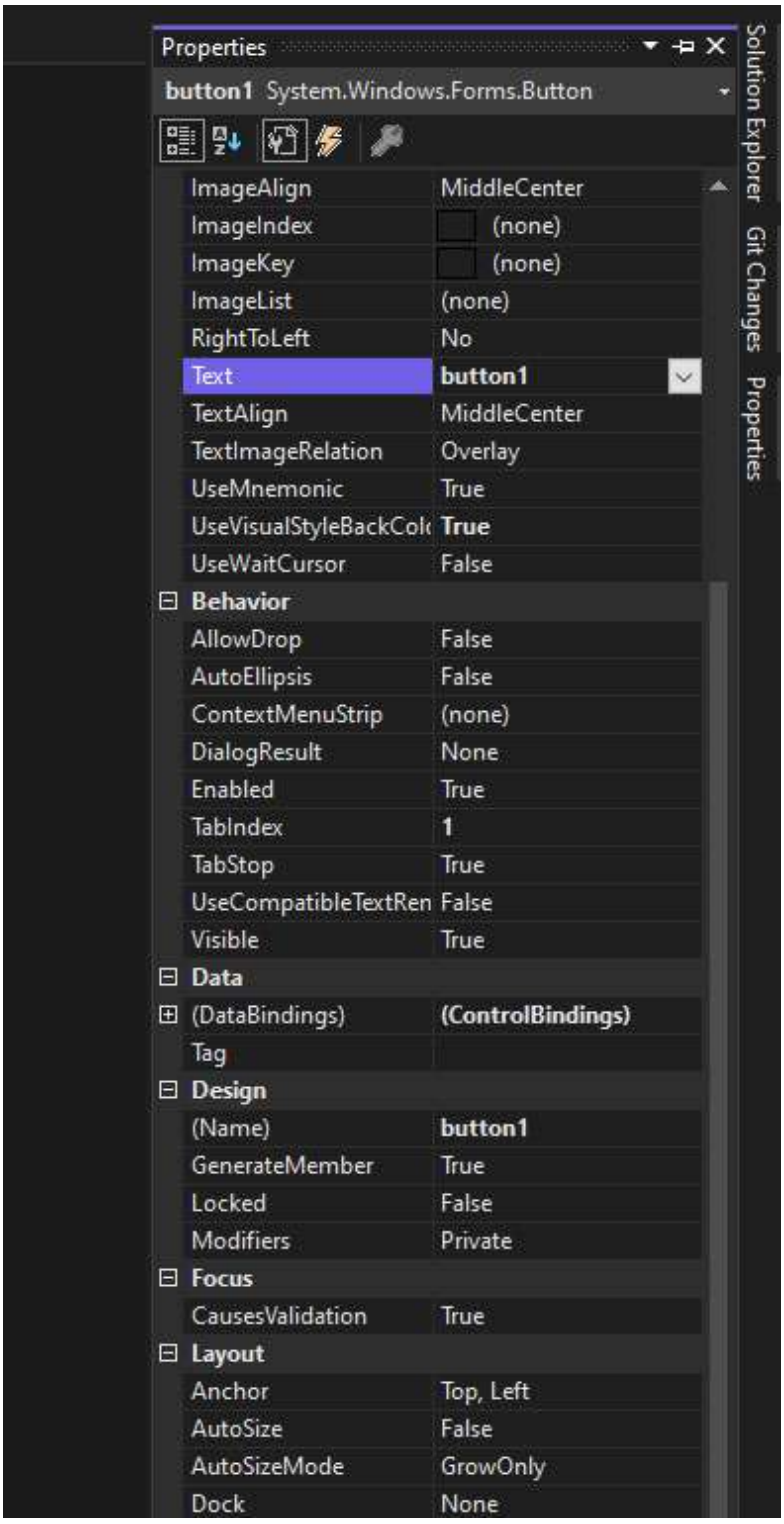


Figure 15.6: *Button component properties*

Each component has relevant properties to change the design, behavior, data associated with it, and layout. The quantity and type of properties vary

according to the component type, but generally, the properties have similar categories. It is recommended that you modify properties for the button or textbox to test the result of each change and the meaning of most of the properties. For example, if you change the background of the button in the properties dialogue, the Design screen automatically reflects the change, as shown in *Figure 15.7*:

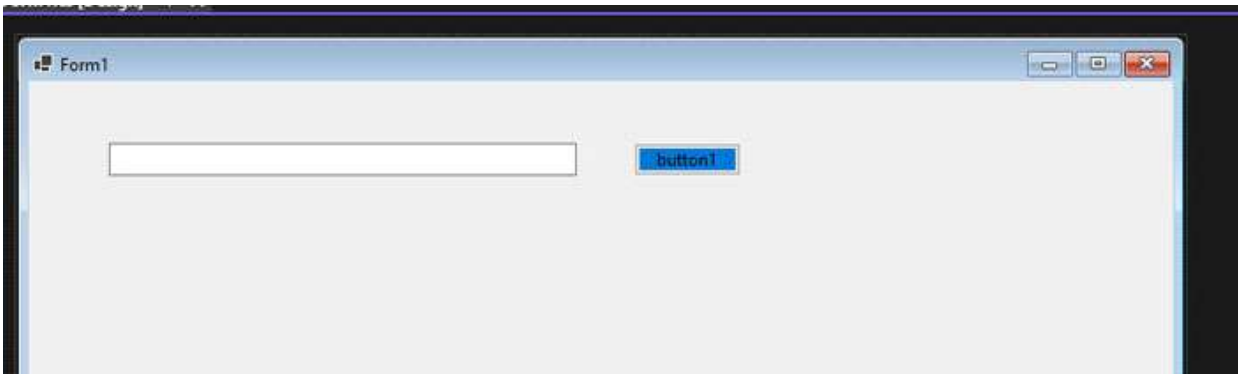


Figure 15.7: *Button background*

After running the planning in Visual Studio, the Windows Forms application will launch, and the design of the application will be identical to the one presented in Visual Studio, as shown in *Figure 15.8*:

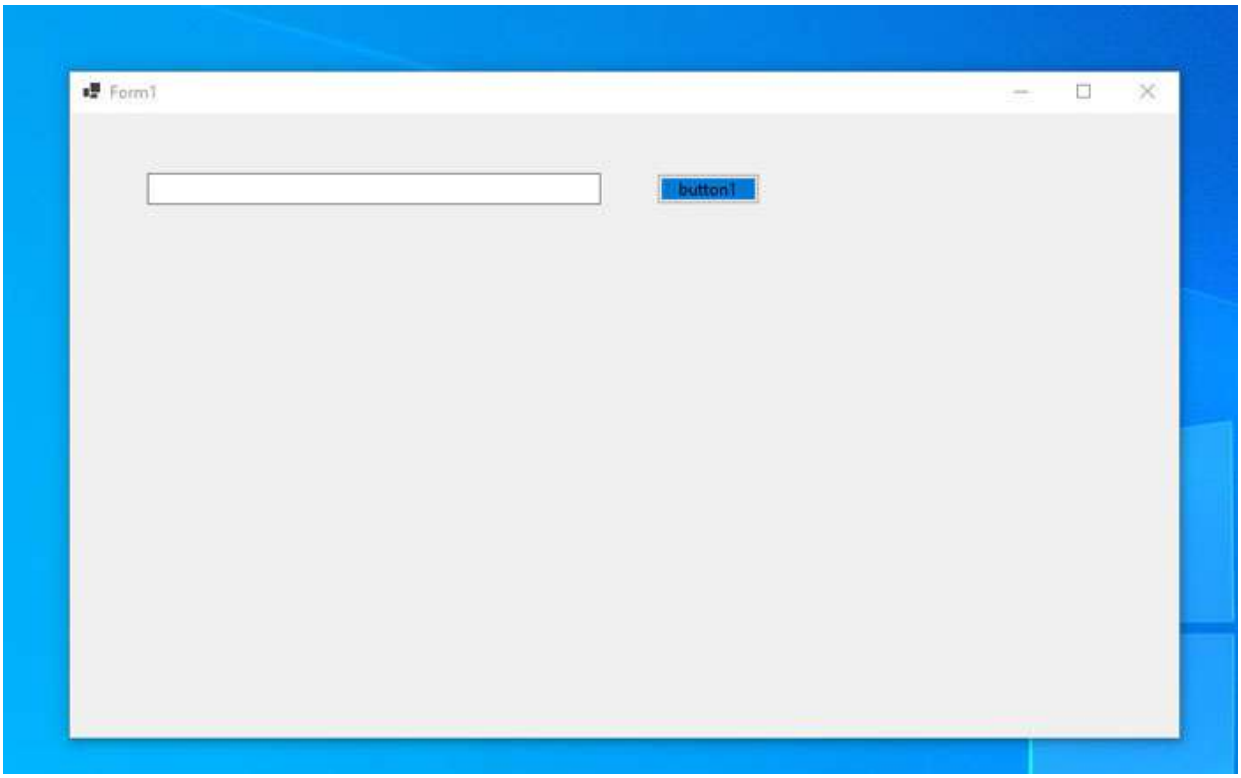


Figure 15.8: *Windows Forms application running*

The compatibility between what is seen in Visual Studio and the actual final result of the application while it is running is one of the things that made Windows Forms so popular, as the result is predictable in terms of design. Compared to Web development, where there are multiple resolutions, web browsers, devices, and other factors that may affect the layout of the pages, there are not many factors that affect the presentation of a Windows Form application in general, making it easier to develop.

However, some factors make developing Windows Forms applications difficult, such as installation, deployment, software updates, compatibility with Linux and macOS, and other factors. The fact that this application is primarily Windows based and requires installation in the user's machine makes the release process a huge challenge in most cases involving a wider range of users and machines.

Additionally, Windows Forms is compatible with almost all the existing libraries in the .NET platform. Therefore, you can take advantage of the Entity Framework and other packages that will allow you to develop an entire enterprise application, connect the application to one or multiple databases, establish integration with other systems, and much more.

Windows Presentation Foundation (WPF)

Windows Presentation Foundation (**WPF**) represented a new generation of frameworks for Desktop development available since 2006, and it was considered a modernization of Windows Forms applications. This new framework is based on XAML as a markup language, allowing the separation of concerns between the user interface and the logical/business implementation, with a similar concept that frameworks for Mobile development have practised. With XAML, it is possible to define styling, rendering data, and defining templates used by multiple screens in the application.

WPF has been used in the market for about 15 years. This type of project became quite mature in terms of features and stability, supported by .NET Core 3.0, .NET 5, and .NET 6, and will be for .NET 7.

As the design process is based on XAML, the adaptation from Windows Forms developers may take a little time in most cases, mainly for complex

applications. However, there is a way to build the screen in Design Mode within Visual Studio, like Windows Forms.

Windows Presentation Foundation allows the developer to build powerful applications combining 2D and 3D graphics, providing a rich experience for the user. It renders faster than the Windows Forms application, representing an evolution of Desktop development within the .NET platform. As it is based on XAML, it has many possibilities for customization for UI, even though the entire customization needs to be done by writing the XAML code from scratch.

To test this kind of project and verify by yourself how it works, open Visual Studio, and within the "Create a New Project" dialog, search for WPF, as shown in *Figure 15.9*:

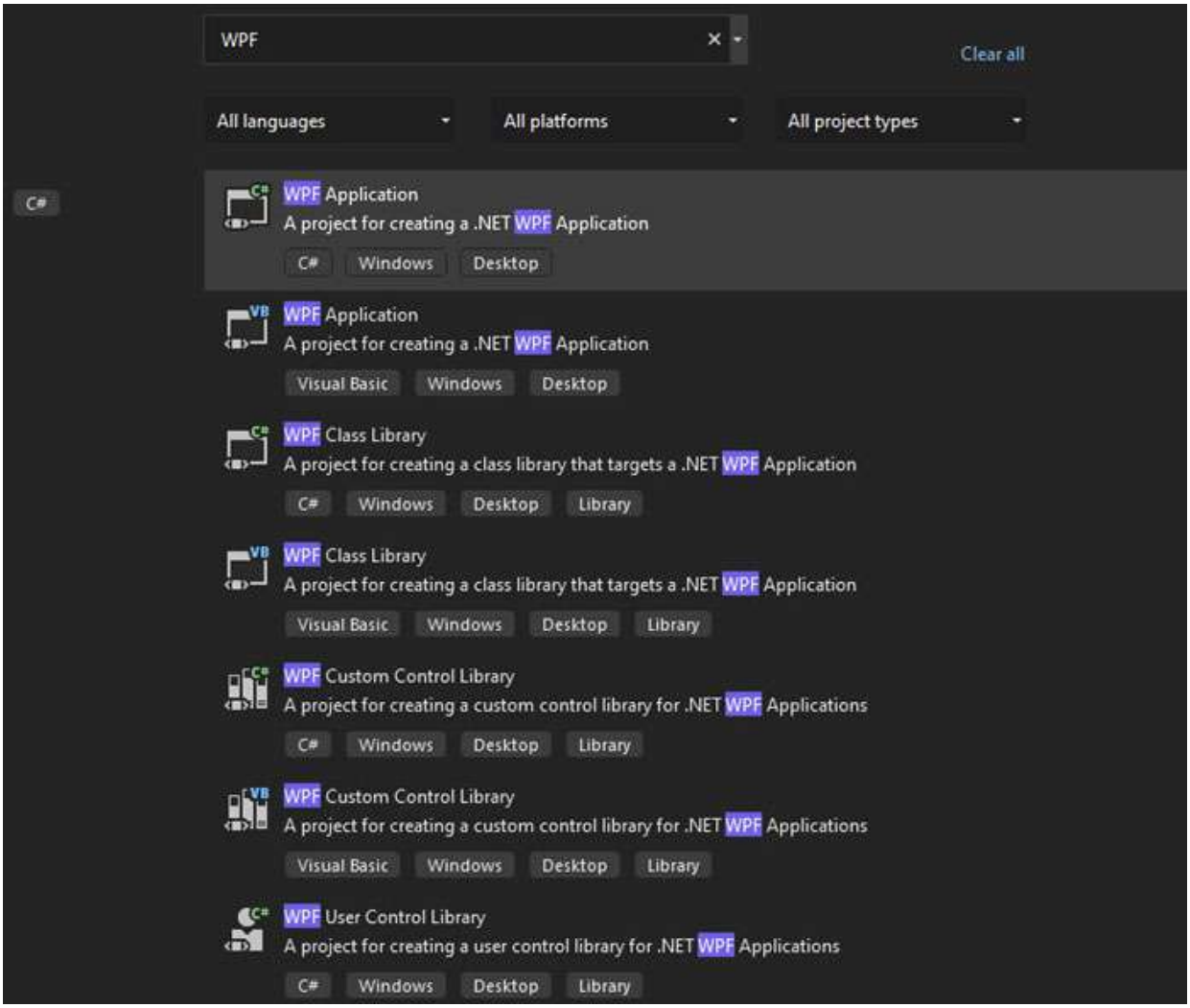


Figure 15.9: *Windows Presentation Foundation project*

- You can choose C# and Visual Basic as the primary language for the project; however, Microsoft has announced that no new features will be added to Visual Basic in feature .NET updates.

After choosing the first project type, give the project a name, and on the next screen, select the .NET version you want to use. For the examples of this chapter, the .NET 6.0 version is used, as shown in *Figure 15.10*:

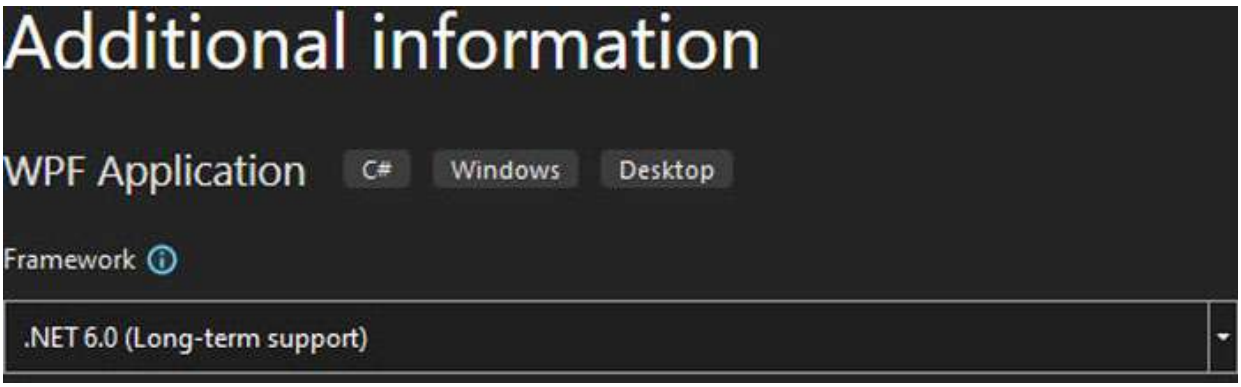


Figure 15.10: *.NET 6,0 for WPF*

After confirming the project creation, the default configuration in Visual Studio shows the Toolbox of components in the left sidebar, the Design representation in the middle, and the underlying XAML code for the Design at the bottom, as shown in *Figure 15.11:*

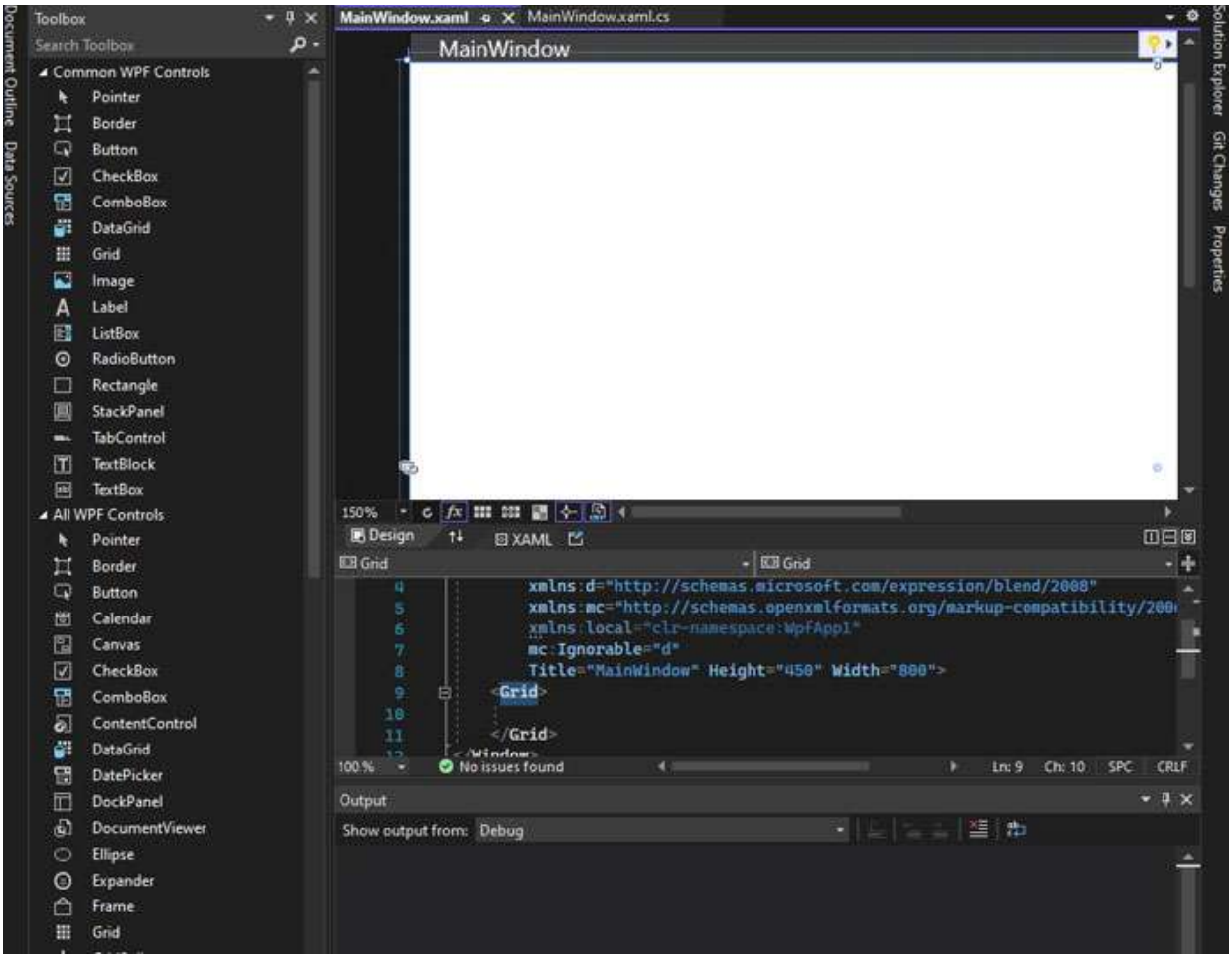


Figure 15.11: *Visual Studio for WPF*

The usability of Visual Studio for the developer is quite similar to the experience of Windows Forms development. However, WPF gives us much more control over the application's design and custom implementation.

After dragging and dropping any component, like a TextBox, the properties for the component can be seen by clicking on the icon at the top of the component, as shown in *Figure 15.12*:

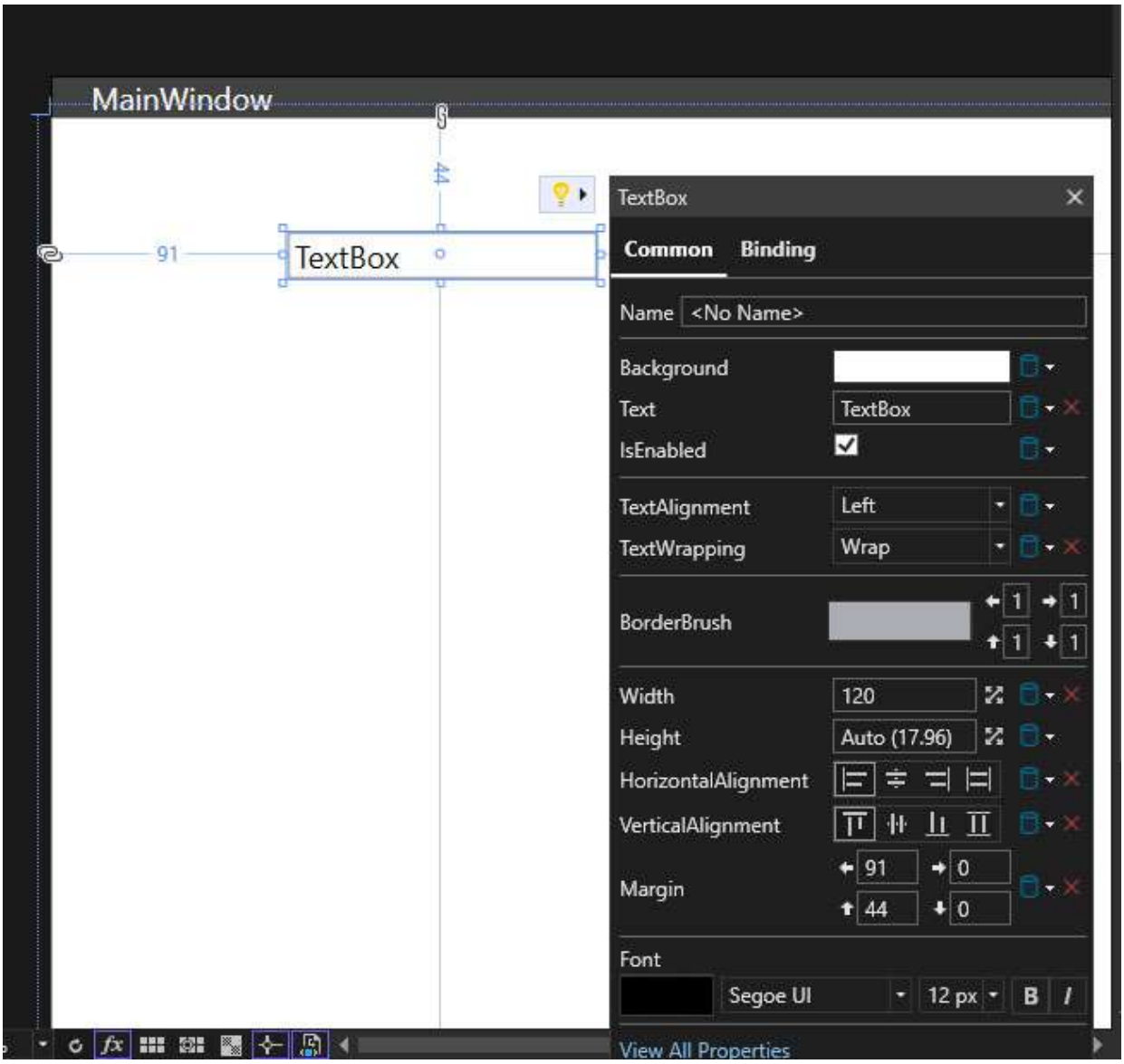


Figure 15.12: *Properties for WPF components*

It is recommended to include more components to test, such as buttons, tables, combobox, checkbox, and other controls, and check all kinds of properties that exist for each component with more details. After including a button and changing the background, text, and size the code looks like the one shown in *Figure 15.13*:

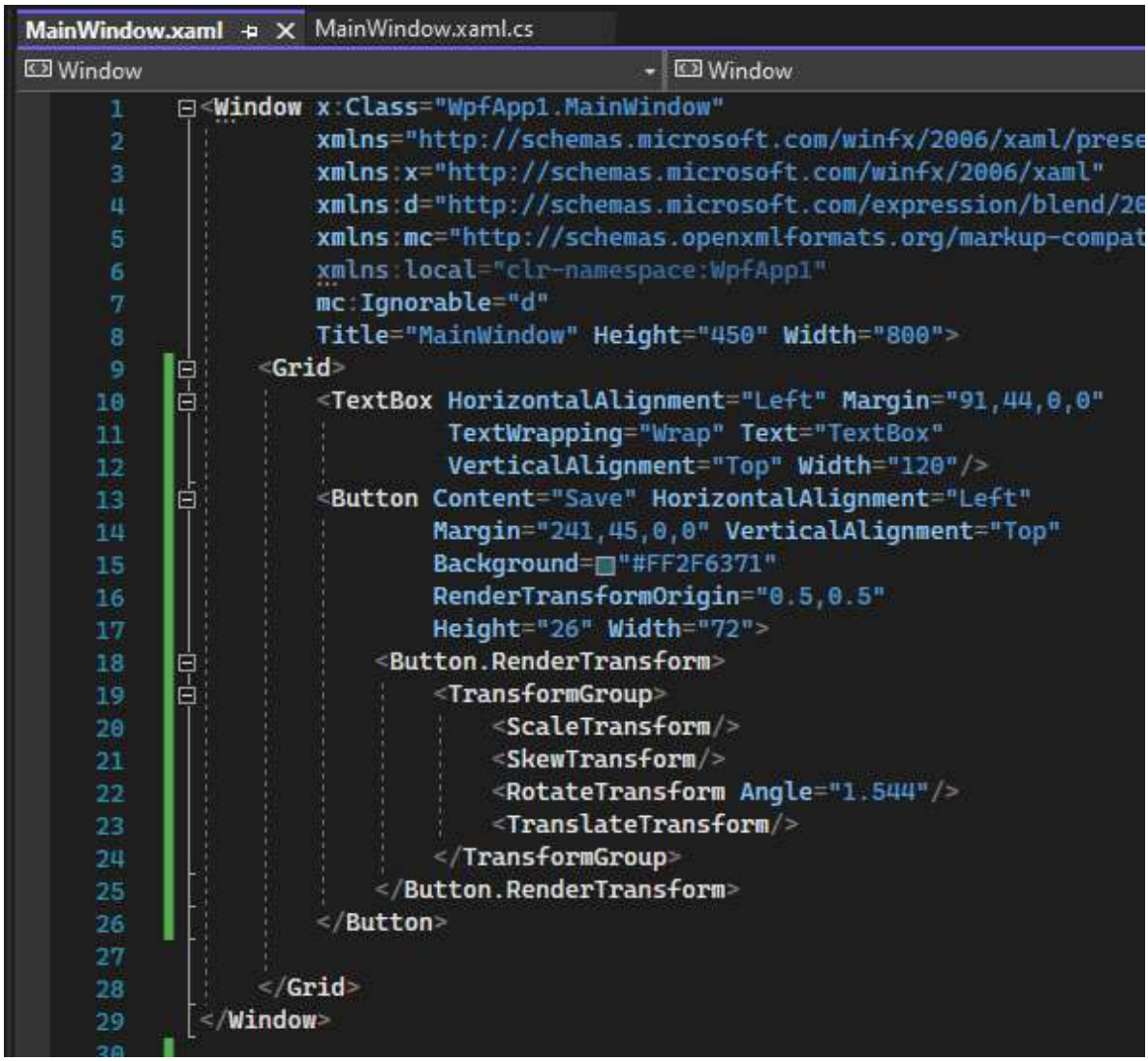


Figure 15.13: *XAML code*

Similar to Web development with HTML, XAML keeps a hierarchical structure for all the visual elements for a WPF screen. As you can see in *Figure 15.13*, the main tag is called “Windows,” and all the other elements are placed within this tag, such as TextBox and Button.

Regarding the individual components, the properties are pretty friendly to understand as they are similar to HTML properties as well, at least in the logic behind them. There are two options to change the properties: in visual Design mode or the XAML. Preferably, most WPF developers tend to manipulate any aspect of design directly on XAML as the Design mode is quite limited and may also affect productivity. Once a developer becomes

familiar with XAML, simple new screens can be quickly developed in minutes.

Each XAML file or WPF screen has the equivalent C# file code associated with it, as shown in *Figure 15.14*:

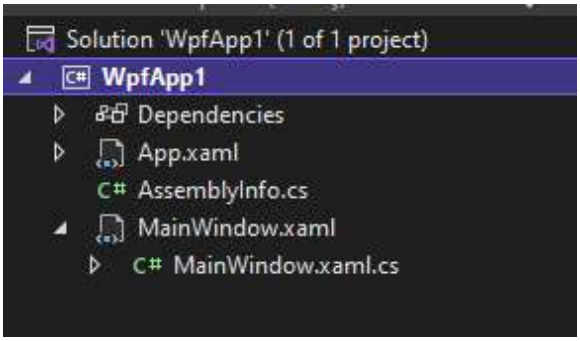


Figure 15.14: *C# file for XAML*

The C# file handles the logical implementation for the WPF screen, such as button-clicking, retrieving data from the database, calling an external API, validating data, and other operations. In Visual mode, double click on the button, and after that, you should be able to see that a **`Button_Click`** method is generated in the C# file for the WPF screen, as shown in *Figure 15.15*:

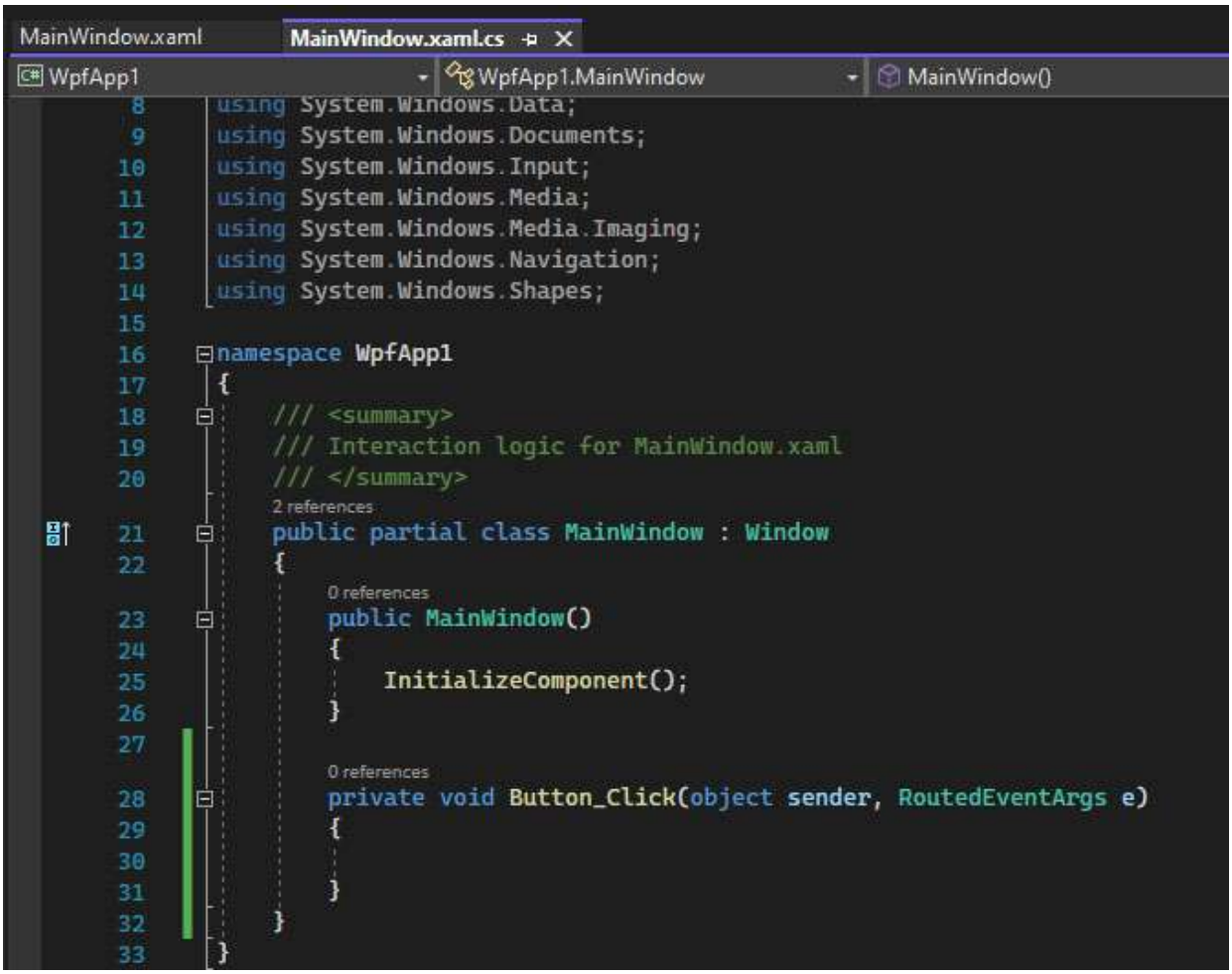


Figure 15.15: *C# file for XAML*

The experience for WPF development is quite similar to Windows Forms in most cases, but WPF has support for all Windows versions and seems to get a huge market compared to Windows Forms over the last 15 years. Additionally, some critical applications built by Microsoft are made in WPF, such as Visual Studio and others, which is a clear indication of huge investments in this technology and a clear case of high performance for complex enterprise applications.

Universal Windows Platform (UWP)

Universal Windows Platform (**UWP**) is the most modern generation for Desktop development within the .NET platform, which the first version was launched in 2016. This new technology also can use XAML to build the

layout. Still, it is adapted to a broader range of inputs and system operating systems that the previous frameworks do not present.

This new framework for Desktop development is only compatible with Windows 10 or later versions and is part of the Windows Runtime API by default. UWP is part of the Windows operating system itself. It receives constant updates throughout Nuget package updates and contains many additional built-in controls for development and design.

Furthermore, UWP applications follow good practices in terms of security and compliance for desktop development as it requires user authorization to access data and device resources from the user's machine.

There are many ways of developing UWP applications. It is possible to use C#, C++, Visual Basic, and JavaScript, including HTML, WinUI, XAML, and DirectX, for development, despite XAML being the most used markup language for this kind of application. Therefore, UWP is more flexible compared to Windows Forms and WPF applications.

UWP was built to have complete integration with Windows 10 and greater versions, giving developers more operating system features. Still, it restricts the number of platforms the application can use as it is not compatible with Windows 8 or any other version before Windows 10. The integration with Windows 10 allows us to publish the application on the Windows Store to monetize your application.

To develop Universal Windows Platform applications using Visual Studio, make sure you have the underlying workload installed, as highlighted in *Figure 15.16*:

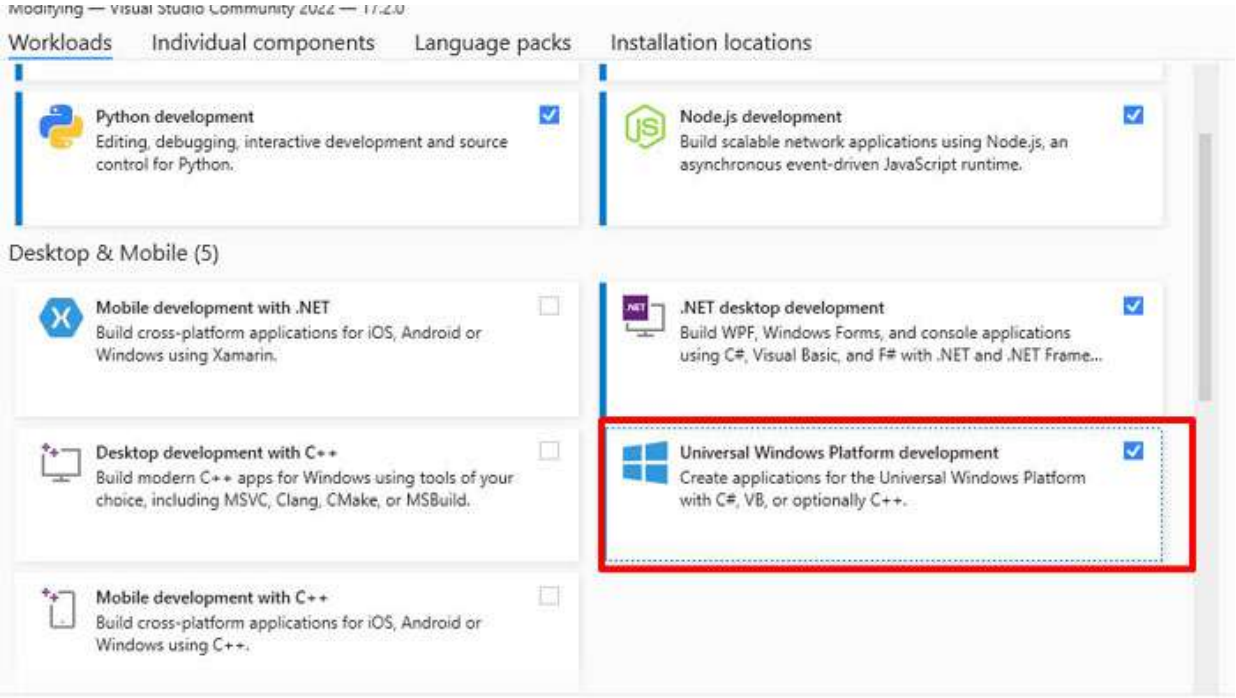


Figure 15.16: *Universal Windows Platform workload*

To create your first UWP application, go to Visual Studio and choose the underlying project template in the dialog box. Depending on your Visual Studio installation, even if you already have the UWP workload, the project template list may warn you to install extra packages. They are two principal project templates for UWP: one based on C# language and one based on Visual Basic. For the examples in this chapter, the C# version of the template is being used, and the project template creation looks like *Figure 15.17*:

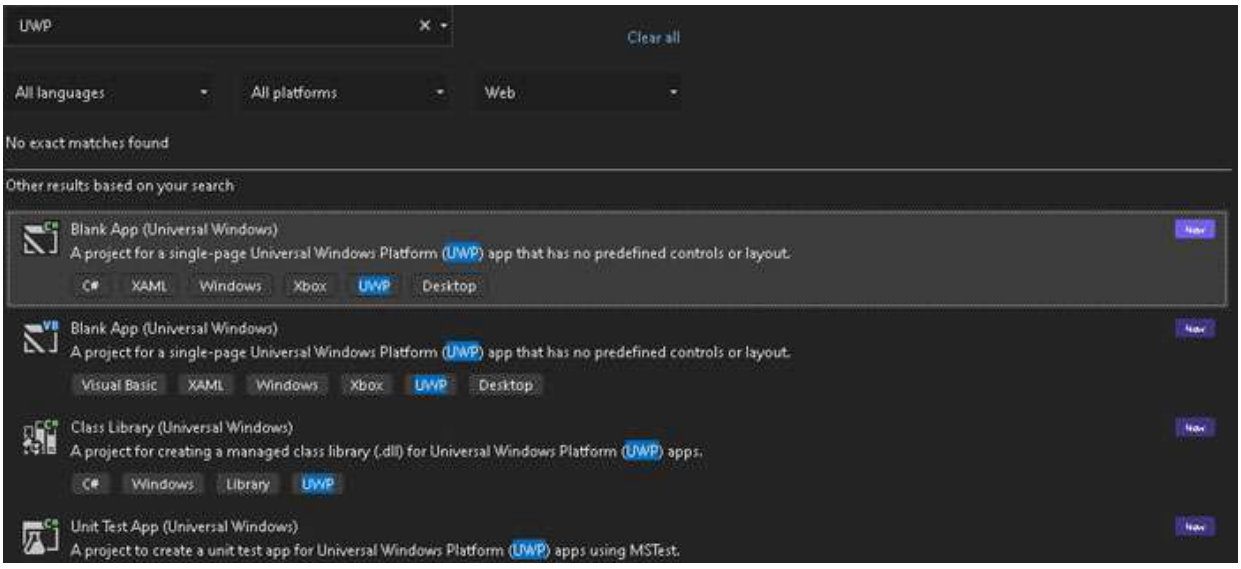


Figure 15.17: *Universal Windows Platform project template*

During the creation of the project, Visual Studio asks which range of Windows versions the app will have support, as shown in *Figure 15.18*:

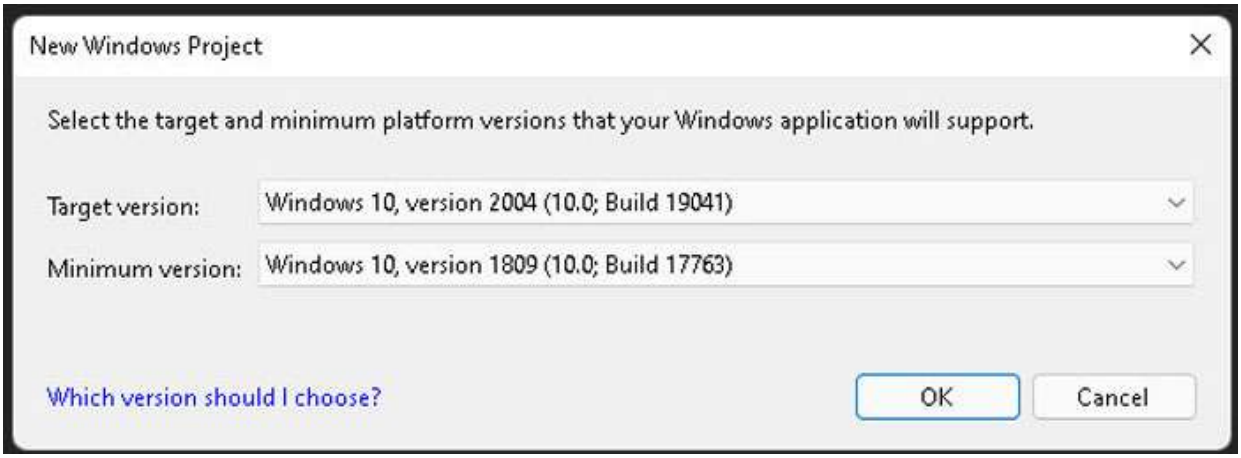


Figure 15.18: *Windows version support for UWP*

Therefore, you can choose specific Windows versions. For the examples in this chapter, the support is from Windows 10, version 1809, to Windows 10, version 2004, but you can choose whatever suits your case. Additionally, when the project is created, Visual Studio automatically opens the Privacy and Security settings for Windows, as the local development of UWP applications requires to have the **Developer Mode** enabled, as shown in *Figure 15.19*:

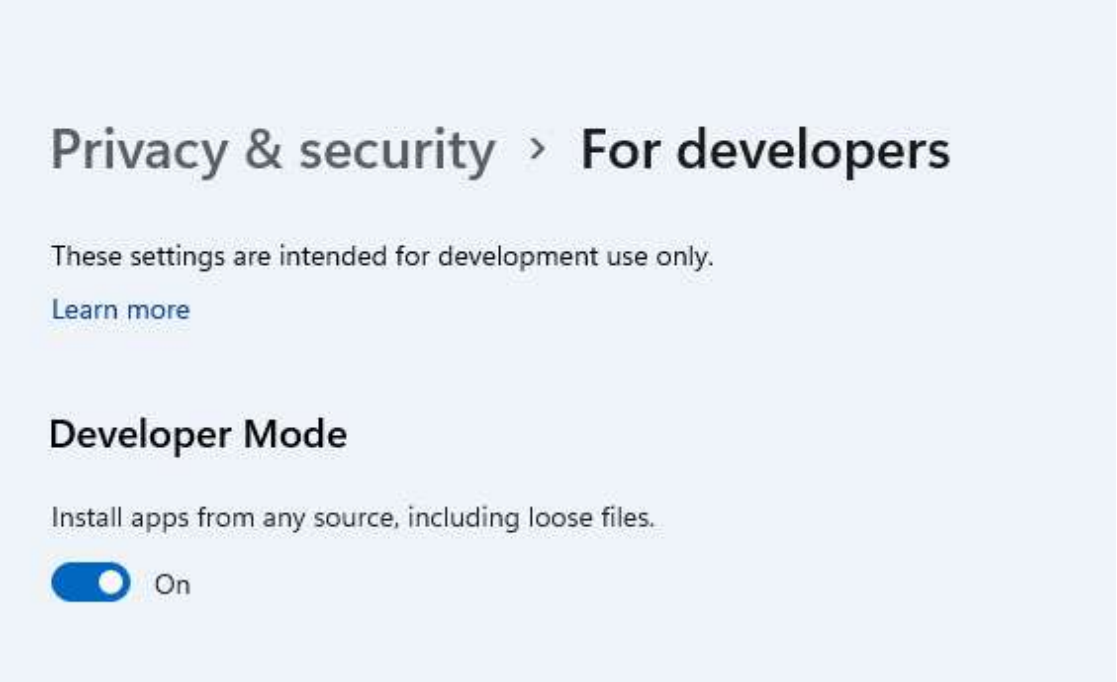


Figure 15.19: *Developer Mode for UWP*

The default project template contains the main XAML file with no layout specification. Once you run the application, Visual Studio shortcuts are shown on the top of the application to facilitate the development, such as opening the XAML file by selecting a specific element on the screen, tracking focused elements, building failure status, and much more, as shown in *Figure 15.20*:

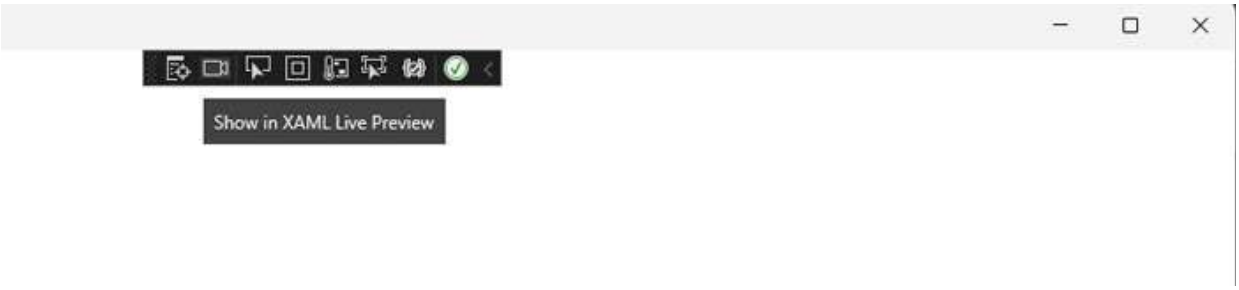


Figure 15.20: *Visual Studio shortcuts for UWP*

As it is possible to use XAML to build the layout of the application, Visual Studio has similar features compared to the WPF development in terms of UI experience within Visual Studio, allowing the developers to use the Design Mode and adapt the properties of any element in the screen, as shown in *Figure 15.21*:

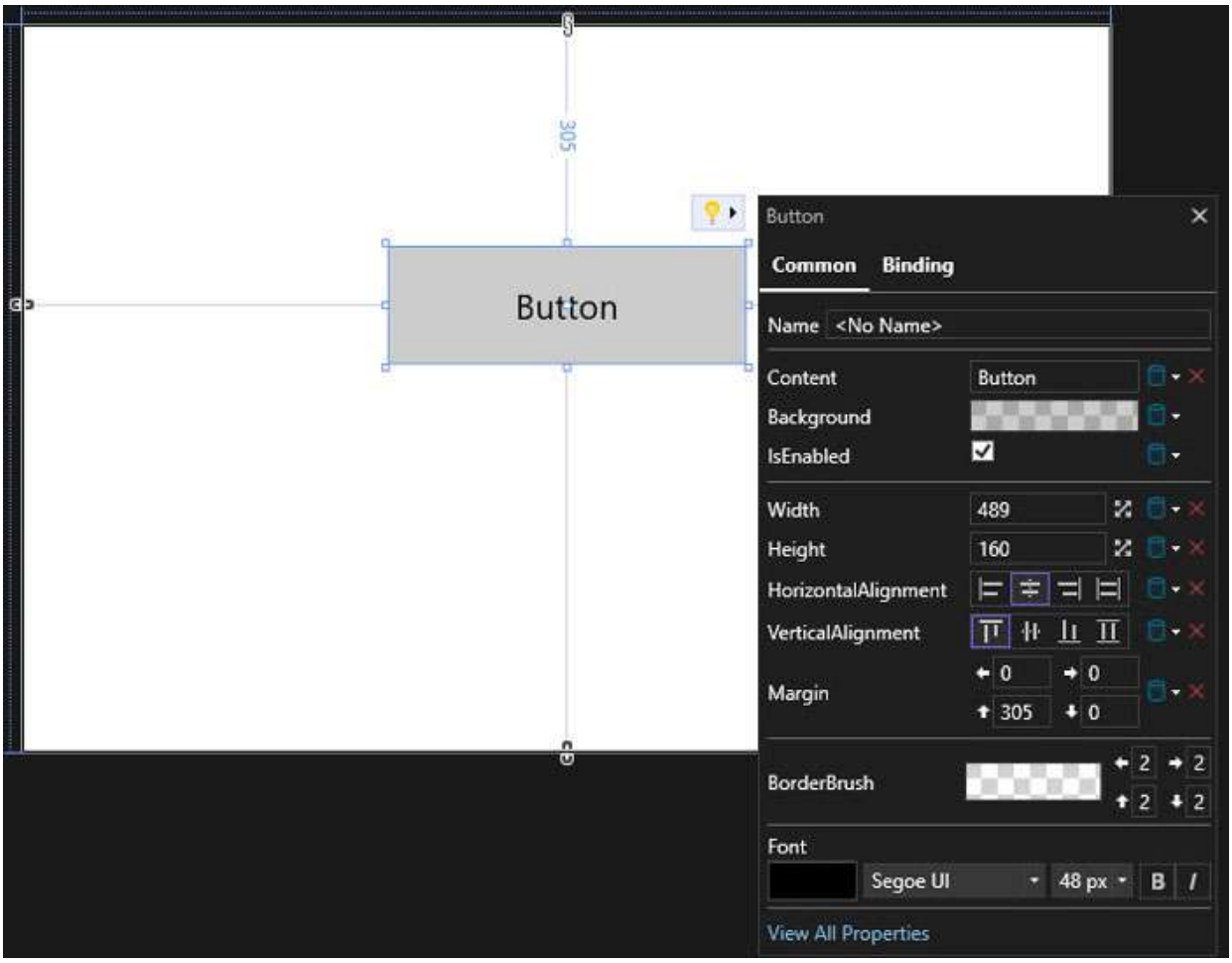


Figure 15.21: *UWP Design Mode*

As Universal Windows Platform applications have access to a broader range of Windows APIs from the user's machine, this kind of application follows a stricter flow in terms of security and privacy than other types of Desktop projects within the .NET platform. It is mandatory to specify a Manifest explicitly pointing out which features of the machine must be installed and run. The same concept is usually applied to mobile development for Android and iOS when applications ask permission from the user to access GPS, Camera, and other features. In the context of UWP applications, if you open the Manifest file presented in the project solution, you will be able to see that there is a Capabilities tab where it is mandatory to specify which features are required from the users that will be installing the application, as shown in *Figure 15.22*:

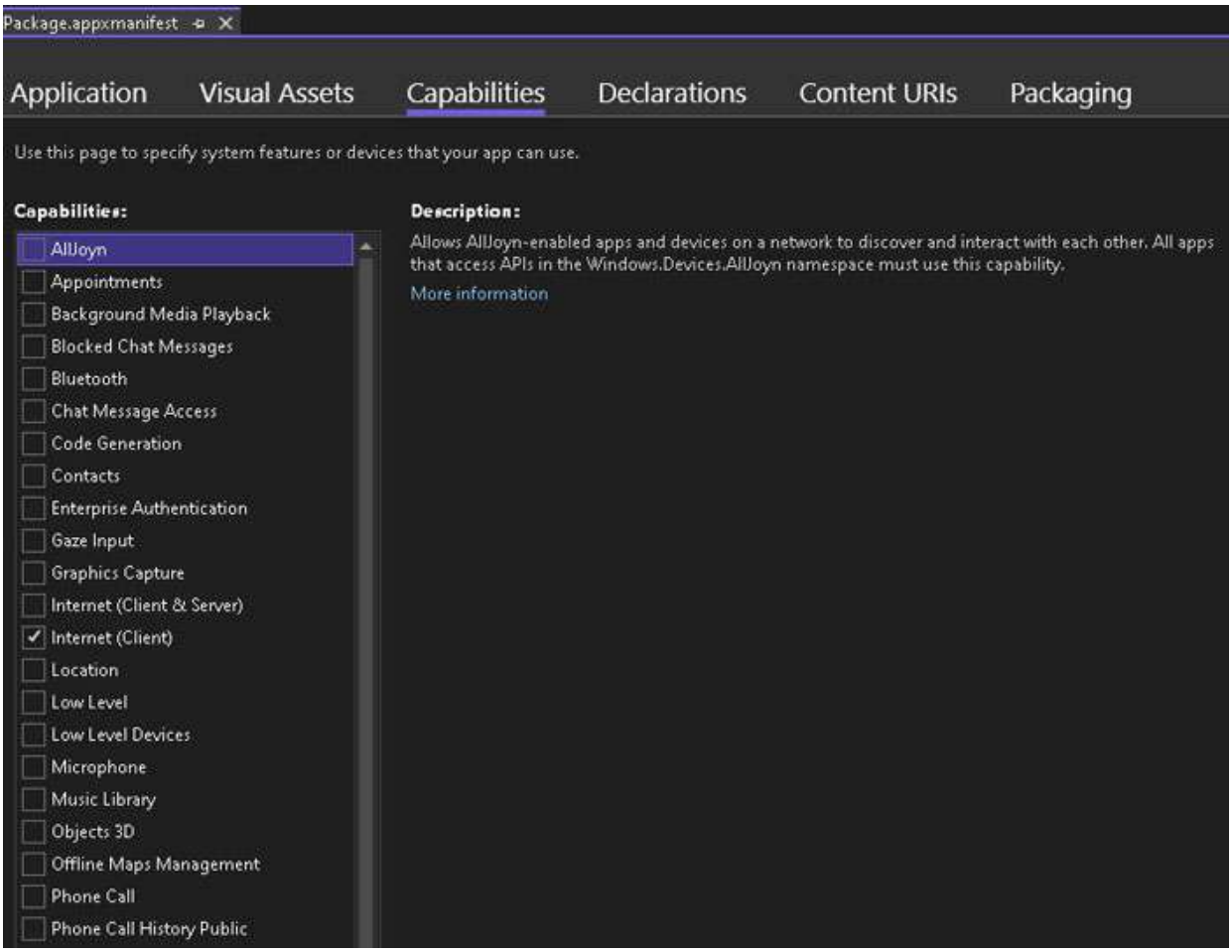


Figure 15.22: *UWP Capabilities requirement*

This flow for installation is expected for any modern application due to security, privacy, and other legal and ethical concerns. Additionally, a detailed manifest helps publish the application in the Windows Store to adequately meet a relevant part of the requirements before allowing users to download and install the application on their machines.

Mobile development

Software development for mobile devices has always represented a considerable challenge in software engineering as there are many types of devices and operating systems, such as Android distributions, iOS, and other platforms. Each device may have its specification, screen resolution, limitations, security restrictions, and obligations in terms of user interface that must be followed for a particular operating system.

In this scenario, multiple programming languages and frameworks can be used to develop mobile applications, such as Java, Objective C, Swift, Flutter, and Ionic, for hybrid applications and much more. It is expensive for any company to keep multiple development teams working on different projects for a different mobile application version for each operating system that only allows a specific programming language.

Given that, there is a relevant problem that software companies, in general, have tried to solve regularly for years, which is related to the attempt to build a single mobile application that can be released to multiple different devices, mainly for Android and iOS devices, including the possibility of using an only one programming language as well. The challenges in terms of compatibility are enormous. Over the years, the puzzle became much more complicated as mobile applications involve cell phones and smart glasses, smart TVs, watches, and many other devices.

The .NET platform has a solid answer for these problems throughout the Xamarin platform, a mature framework for cross-platform mobile development, which gives us the possibility of developing applications for Android and iOS using C# language, and the framework itself takes care of binding the compatible version for the various operating systems.

To start with Xamarin development, you must have the underlying workload installed in Visual Studio with the Mobile option development with .NET, as shown in *Figure 15.23*:

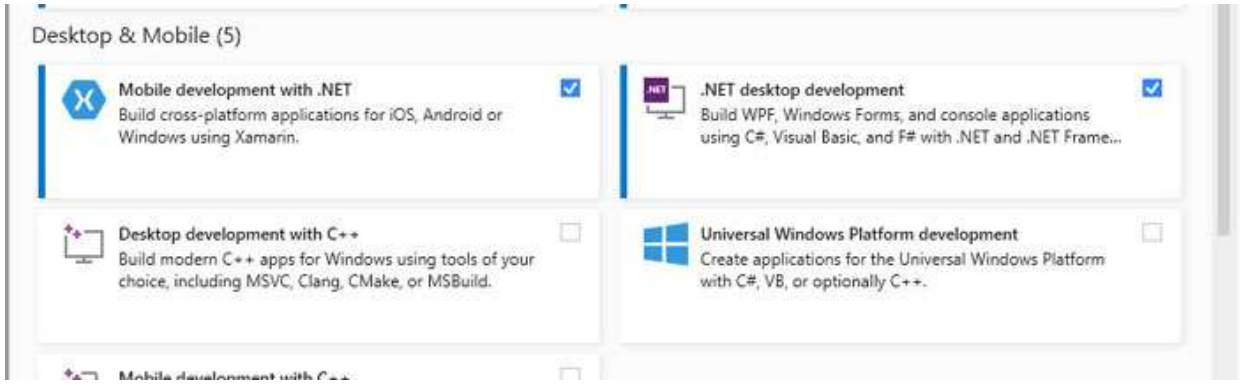


Figure 15.23: *Mobile development workload*

Xamarin became a robust framework for multi-platform development over the years, allowing applications for cell phones based on Android and iOS operating systems and the development of applications for smartwatches, TVs, and many other devices.

When creating projects from scratch using Visual Studio, it is possible to choose project templates for a specific operating system, certain device types, or a more generic Xamarin project template that would support multi-platform development and deployment, as shown in *Figure 15.24*:

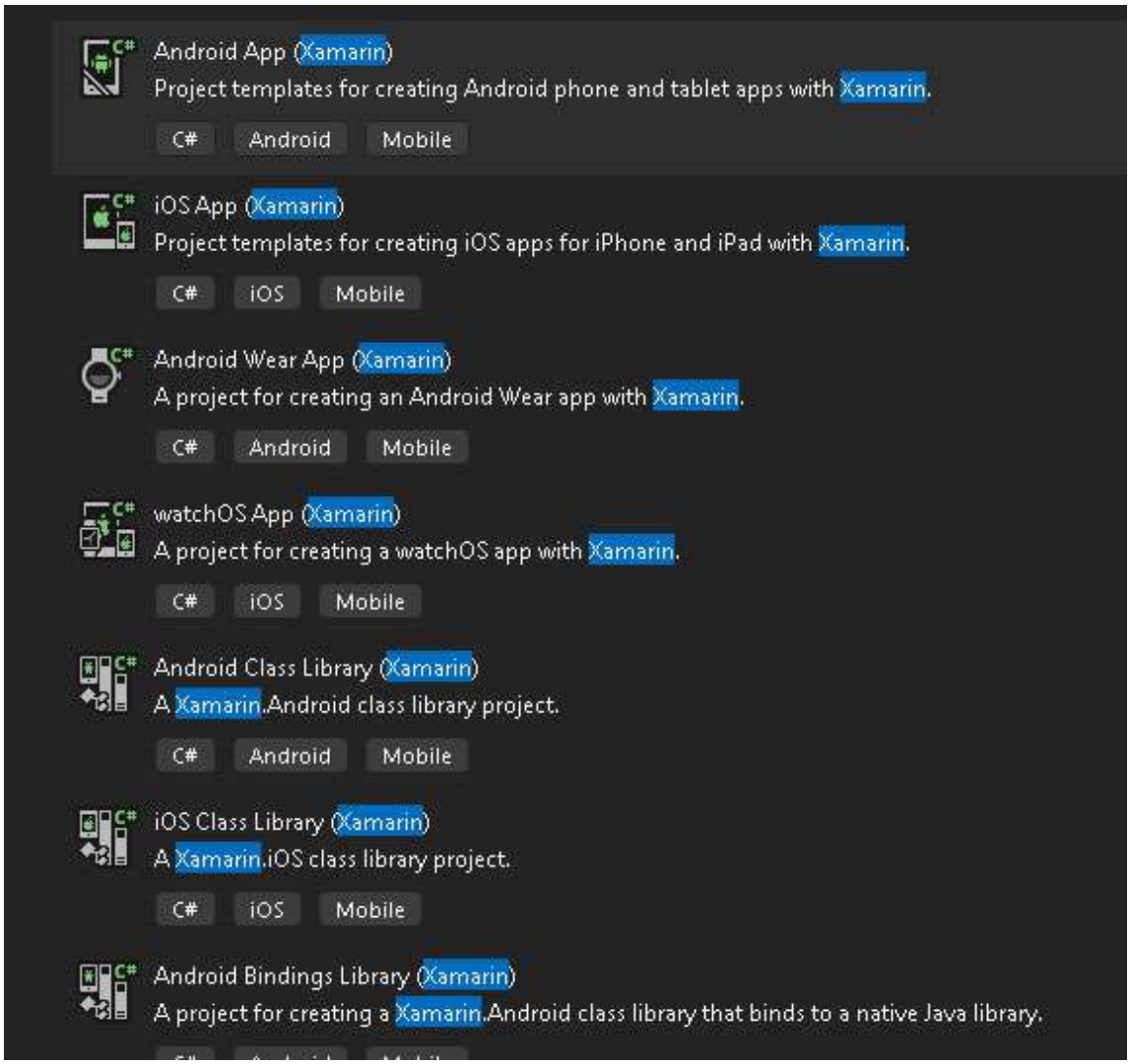


Figure 15.24: *Mobile development workload*

To create your first Xamarin App, choose the option **`Xamarin.Forms`**, highlighted in *Figure 15.25*:

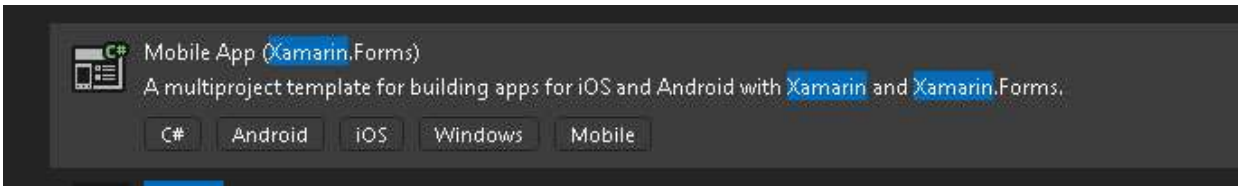


Figure 15.25: *Mobile App Xamarin Forms*

After giving the project a name, Visual Studio asks what type of template **`Xamarin.Forms`** should apply to the project. There are three main options: Flyout, Tabbed, and Blank. In the context of the examples of this book, the Flyout is being applied. Additionally, there is a possibility to choose which platforms the new app will have support: Android, iOS, and UWP. Choose the option Android and iOS, as shown in *Figure 15.26*:

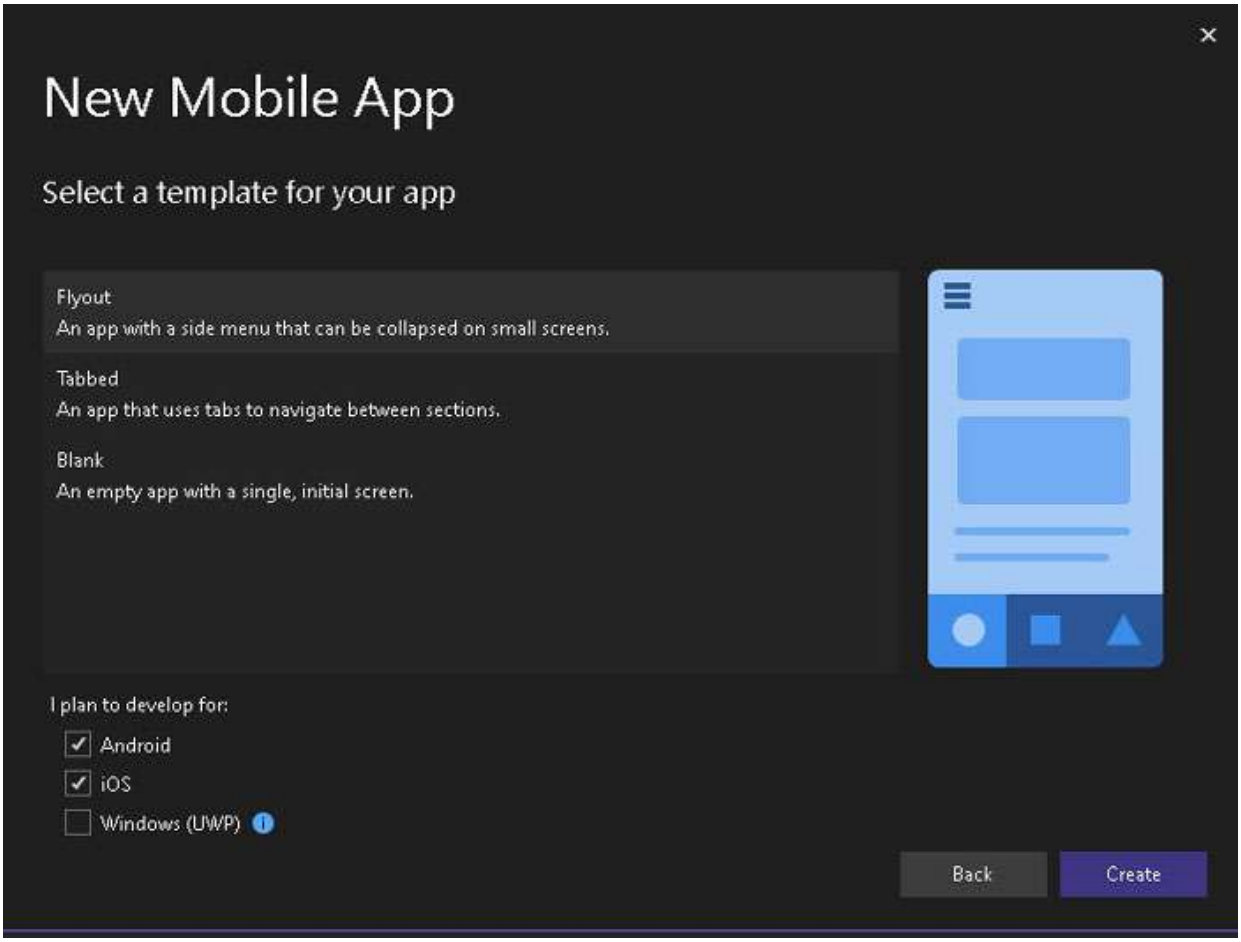


Figure 15.26: *New Mobile App Configuration*

After creating the application, the Solution Explorer in Visual Studio shows the files and folder structure for the application, consisting of different projects for Android and iOS and one general App project that contains the code usually shared across the two other projects.

Note: This section aims to show you the types of projects available to create mobile applications within the .NET platform, exploring the

most high-level details of each option. However, we recommend you check the official Xamarin documentation on the underlying Microsoft website for more detailed information, as this framework is constantly changing and evolving.

The Solution Explorer should look like the representation in *Figure 15.27*:

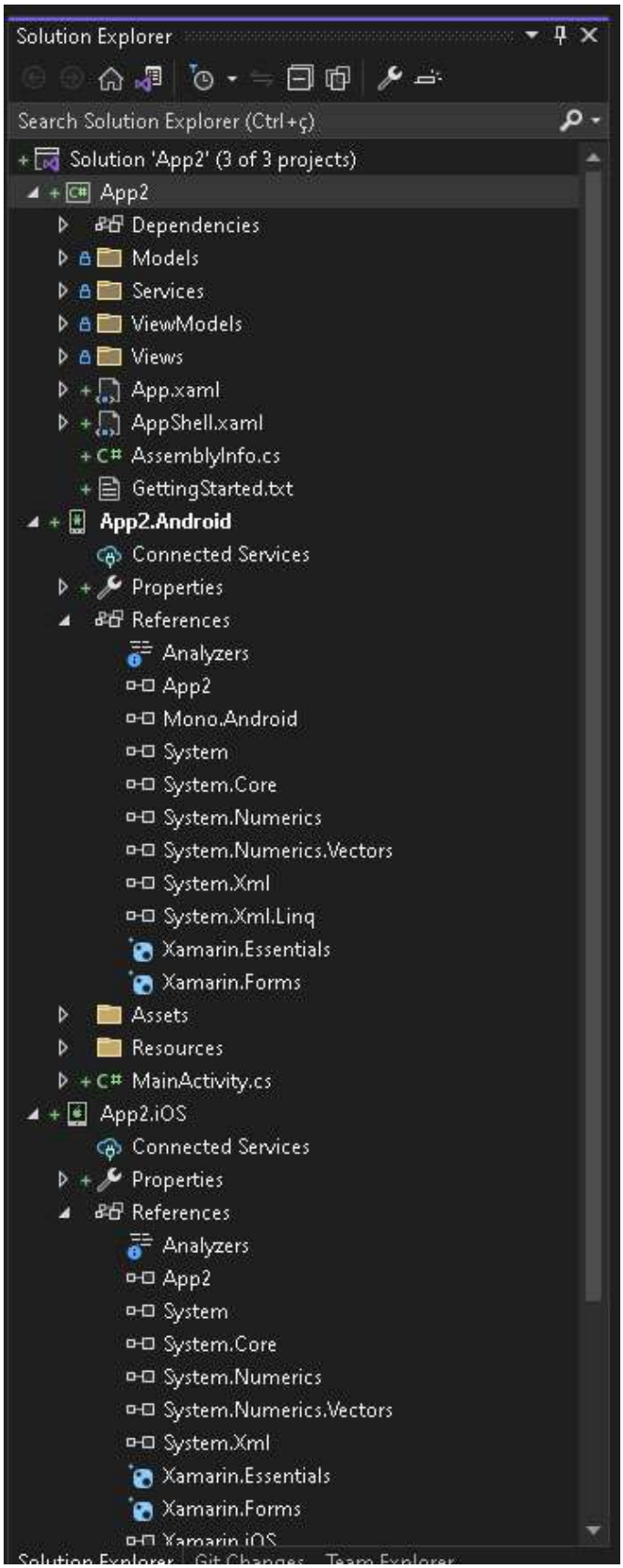


Figure 15.27: *Solution Explorer for Xamarin.Forms*

This separation of concerns based on the platform is important to manage each platform's custom implementation, including distinct layout definitions and much more. The layout definition is based on XAML, already shown in this chapter. Therefore, if you are familiar with XAML, you can seep up on developing WPF, UWP, and Xamarin applications.

To emulate the application on mobile devices, you can use the default Android emulator presented on Visual Studio, install new emulators, or even run it directly on your phone plugged into the PC. You can choose an emulator, as shown in *Figure 15.28*:

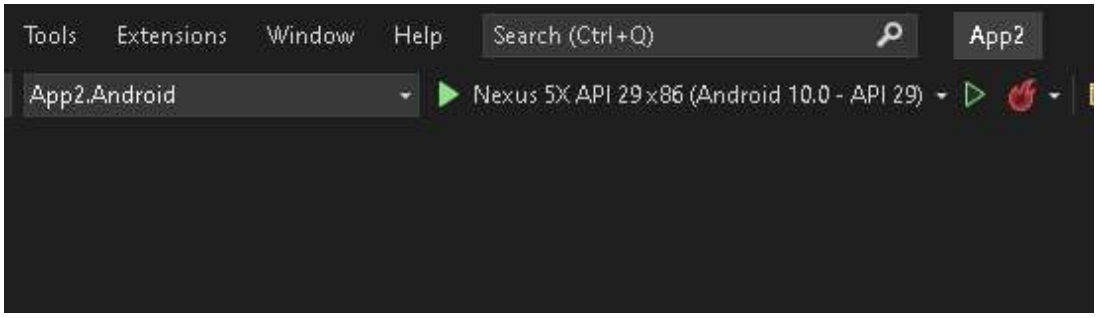


Figure 15.28: *Android emulator*

In the context of this chapter, the Nexus 5X emulator is being used with Android 10.0. Furthermore, for both Android and iOS development, they share the same way of building layouts using XAML, and both of them use C# language for logical and business implementation, as shown in *Figure 15.29*:

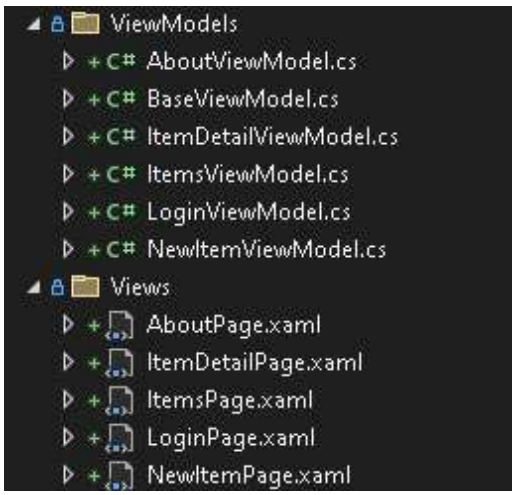


Figure 15.29: *Xamarin files for layout and logic*

C# and XAML in multi-platform development represent a considerable advantage. There is no need to learn multiple languages, such as Java, Objective C, and others, to build applications for different operating systems and many device options. The Xamarin framework makes the underlying deployable app for each platform. It is becoming more mature regarding compatibility and adaptability for native Android and iOS application development.

Console applications

Console applications in the .NET platform represent a popular option for those learning the C# language. It does not require building interfaces. It is not obligated to set up any extra configuration just to run the application in the default project template.

Console applications represent an excellent option in specific scenarios where there is no need for a robust User Interface or even cases where there is no need for user interaction with the application. It is the simplest project type of the .NET platform.

To create your first Console App, go to Visual Studio and choose the underlying project template target to C# language, as shown in *Figure 15.30*:

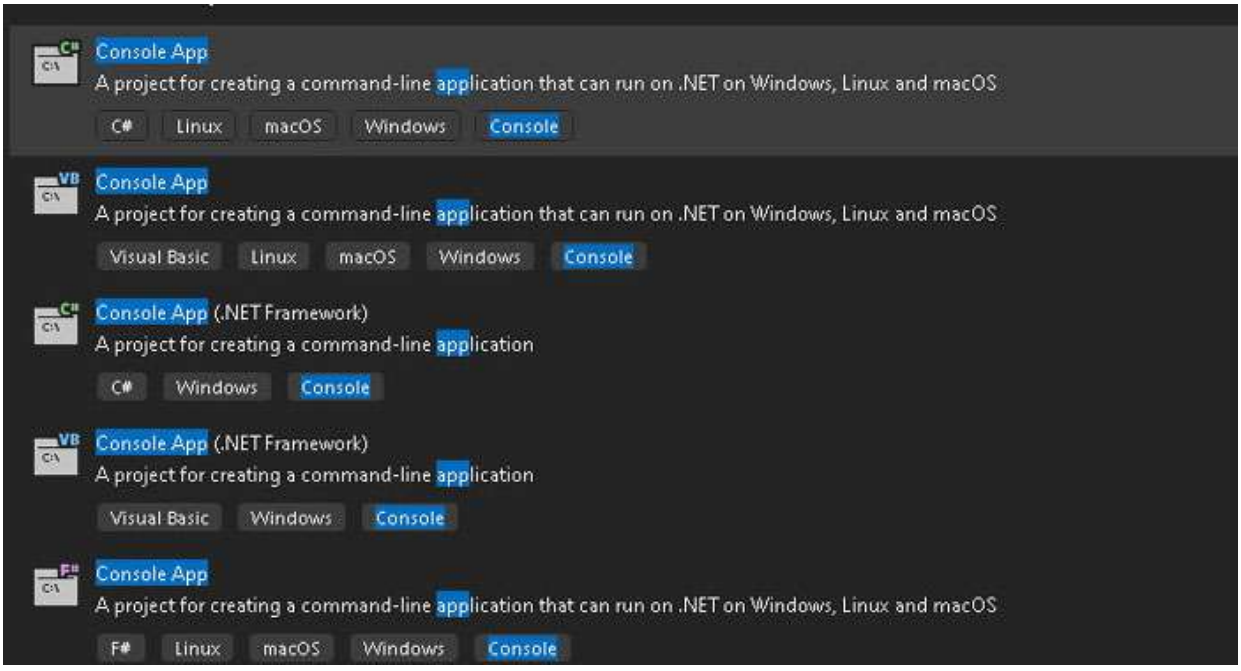


Figure 15.30: *Console App project template*

The Visual Studio also contains the same template for other programming languages within the .NET platform, such as Visual Basic and F#. However, C# is the programming language chosen for the example of this chapter.

After confirming the application's creation, you will see the file structure in the Solution Explorer and realize that the application primarily has a single file called **`Program.cs`**. With Visual Studio 2022 and C# 10, the Console App is simplified compared to previous versions. There is no default method in the **`Program.cs`** file and default namespace references. You can start coding your application directly and run it, similar to what happens with Python and script languages, as shown in *Figure 15.31*:

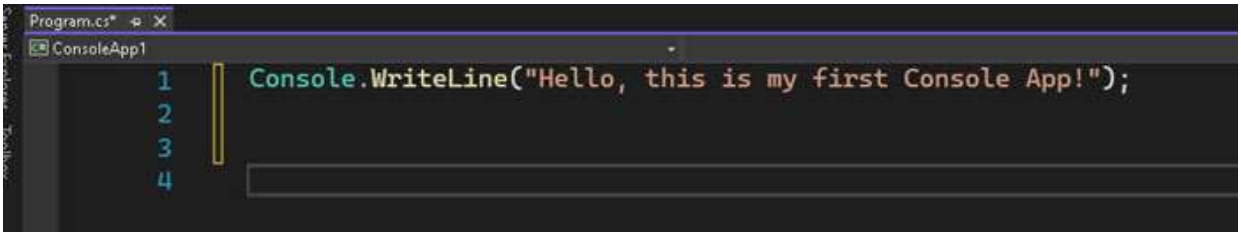


Figure 15.31: *Console app*

If you run the application, a default black screen application will be started, and the message will be printed in the console, as shown in *Figure 15.32*:

Figure 15.32: *Console app running*

Console Apps are usually used to run Jobs, including cases where there is a need for running background tasks on servers. Additionally, this type of project is suitable for C# learners. You can create classes, interact with the database, use any library available in the .NET framework and see the underlying output.

Conclusion

In this chapter, you had an overview of the most relevant project types within the .NET platform for Desktop and Mobile development, understanding the difference between all the generations of project types for Desktop applications and the options for Mobile development.

The .NET platform constantly modernizes all the available project types after each annual version. The development of Desktop and Mobile applications in .NET and C# followed the most significant trends in the market over the years, adapting the platform to allow the deployment of fast and robust enterprise applications that represent an alternative to mobile development in many scenarios.

In the next chapter, you will have the opportunity to learn the most significant features in .NET for cloud development combined with Azure resources.

Points to remember

- Windows Forms and most project types for Desktop applications in .NET are not compatible with Linux and macOS.
- Windows Forms is the precursor of all the frameworks for desktop development within the .NET platform.
- .NET Core 3.0, .NET 5, and .NET 6 versions fully support Windows Forms, WPF, and UWP applications.
- WPF and UWP use XAML as a base markup language for the layout and design aspects of the application.

Multiple-choice questions

1. **Which alternative represents an advantage of Desktop development compared to Web development?**

    a. Easier deployment

    b. Scalable

    c. Possibility of access to more features from the user’s machine

    d. Facility to upgrade the software

2. **Which markup language can be used to design WPF applications?**

a. HTML
b. CSS
c. XML
d. XAML

Answers

1. **c**
2. **d**

Questions

1. Explain in which scenarios the development of Desktop applications would be suitable.
2. Create a WPF application from scratch containing a Grid and search capabilities using what you have learned from this chapter.
3. Explain the differences between Windows Forms and WPF technologies.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 16

Azure Integration Services

Introduction

Modern software development involves knowledge of cloud services and architecture patterns suitable for distributed systems, allowing developers to create an application that would be scalable, reliable, and cost-effective in terms of infrastructure. The .NET platform offers a wide range of libraries and packages to facilitate the integration with Azure cloud services, including possibilities of simulating cloud infrastructure locally.

Learning what the .NET platform supports for cloud development for Azure will allow you to make proper technical decisions for projects that would be more suitable to be hosted in cloud services, allowing you to take advantage of applying serverless and distributed systems concepts.

This chapter will cover the integration with the most relevant resources on Azure for .NET applications, such as Blob Storage, Azure Functions, Azure App Service, and Azure App Configuration.

Structure

In this chapter, we will discuss the following topics:

- Azure storage accounts
- Creating an Azure storage account
- Azure functions

Objectives

After studying this unit, you should be able to understand the most relevant resources on Azure to build Web applications, integrate basic .NET applications with Azure services using the underlying packages for C# language, and understand basic concepts of serverless and cloud development concepts.

Azure storage accounts

Most web applications need to store images, static files, and content from different formats to allow users to upload content to the application server. The same applies to Console applications that need to run file-processing jobs and Desktop applications that allow file upload functionality.

There are many ways of storing these files such as:

- Store these files in a folder within the application server
- Have a dedicated machine to store the files
- Store the files in folders among the application files
- Store them in the database as blobs
- Use a storage system to store the files outside the application context

All these options have pros and cons depending on the context of the technical and non-functional requirements an application must have, but in general, there are good practices in terms of file uploads that are recommended to follow for any project. One of these good practices involves the separation of user content files from the application server into a separate storage service, which allows a more efficient way to provide files (bandwidth) and has high capabilities for data replication, security policies, and automatic backup tools.

Considering these practices, Azure Storage Account represents a great alternative in terms of scalability, security, performance, and costs; to be able to use this cloud resource with .NET applications as Microsoft has provided packages to speed up the development, even though it is possible to do any integration with Azure using REST APIs as well.

Creating an Azure storage account

Microsoft has programs for IT students, partnerships with universities, and policies around new Azure accounts that allow us to test the creation and execution of specific types of resources on Azure without costs for a certain period. To know more about the free services available, please visit the official Website at **https://azure.microsoft.com**.

To start using Azure Storage accounts, you must first register into Azure Portal under the official website. After this, go to the "**Create a resource**"

option on the portal, as shown in *Figure 16.1*:

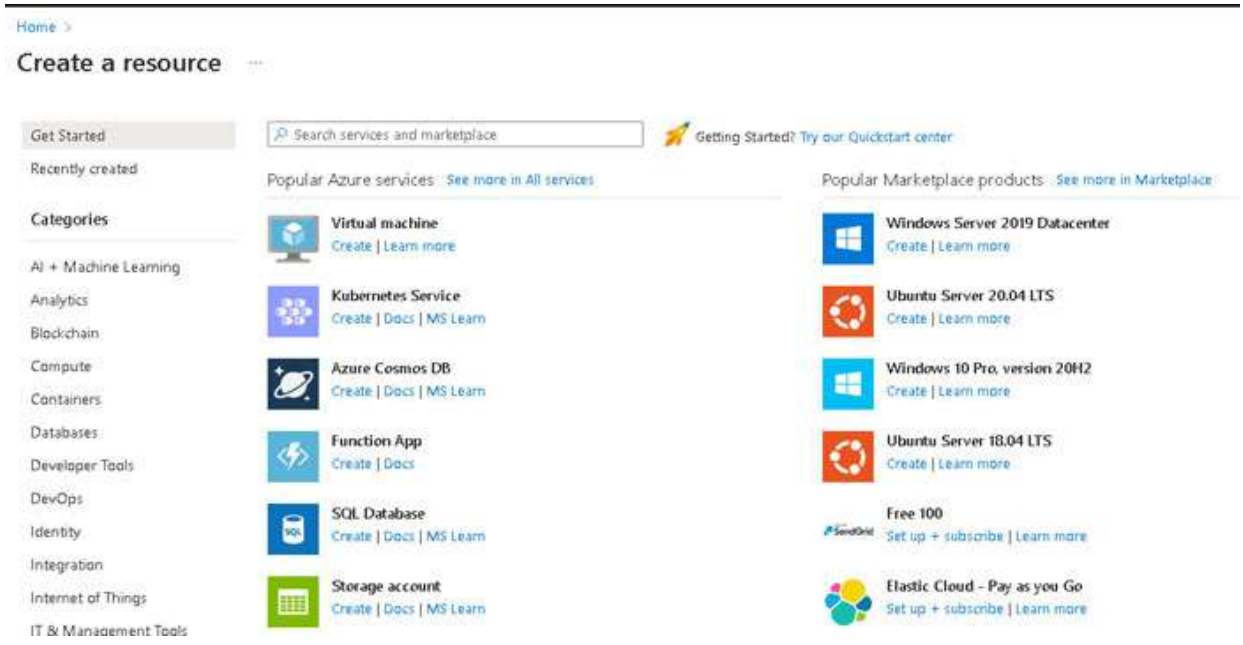


Figure 16.1: *Azure resources*

At this point on Azure, the resources are separated by category based on the type of use, such as Containers, Databases, Developer Tools, Develops, the Internet of Things, and much more. Choose the Storage account option, and specify a subscription, resource group, region, and redundancy type, as shown in *Figure 16.2*:

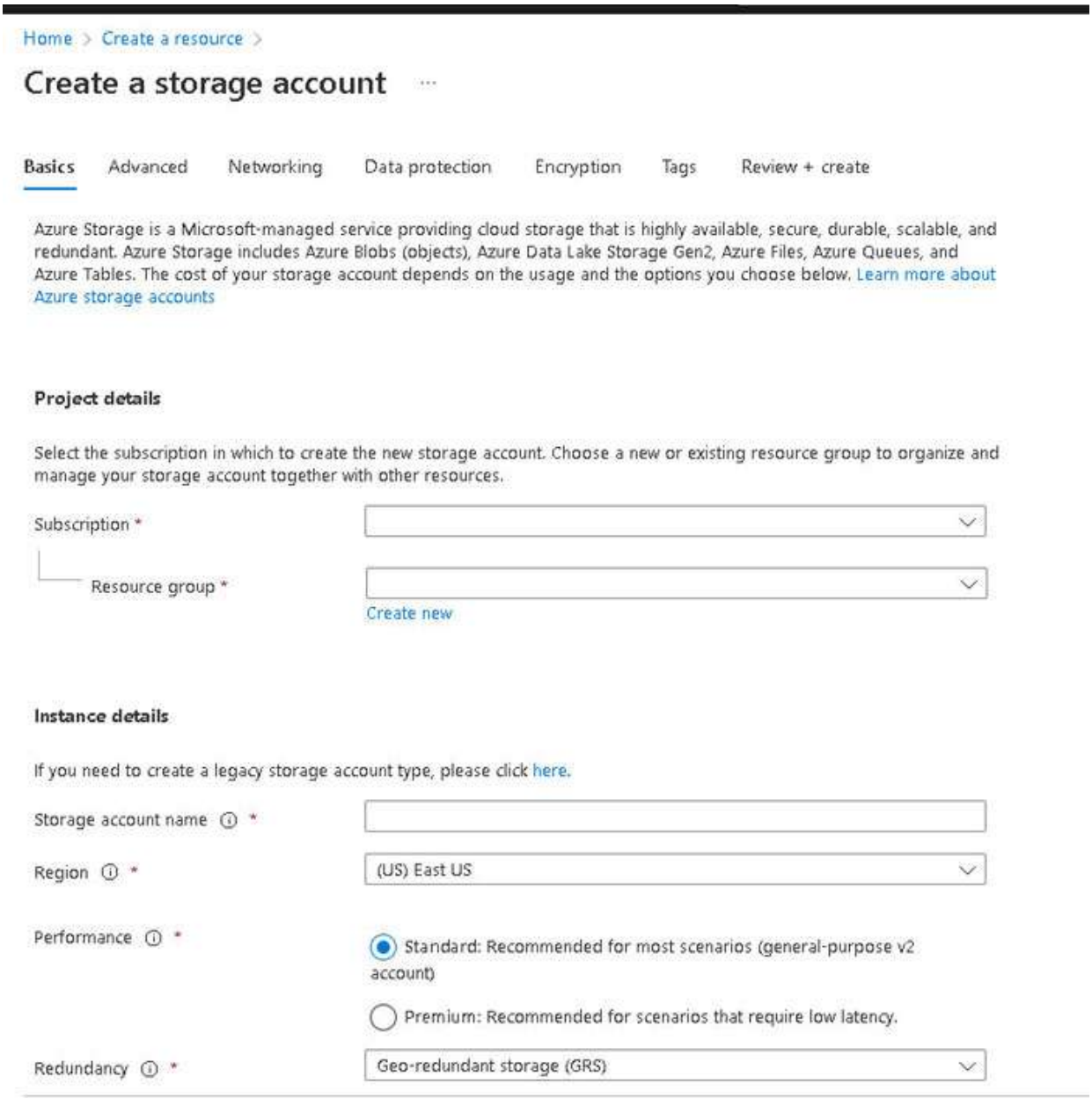


Figure 16.2: *Storage account details*

Since storage accounts on Azure use public endpoints with a subdomain specified by Azure by default, the storage account name field must be filled with a unique name, not only in terms of your resources but also considering all the storage accounts created globally by other tenants. After specifying a unique name, it is possible to specify a detailed configuration for Networking, Encryption, and Tags. In the context of the example of this chapter, all the default configurations are being used. The "**`Review +`**

`Create`” option can be used to check the default configurations. As shown in *Figure 16.3*, a “**`dotnetbook`**” storage account name is specified:

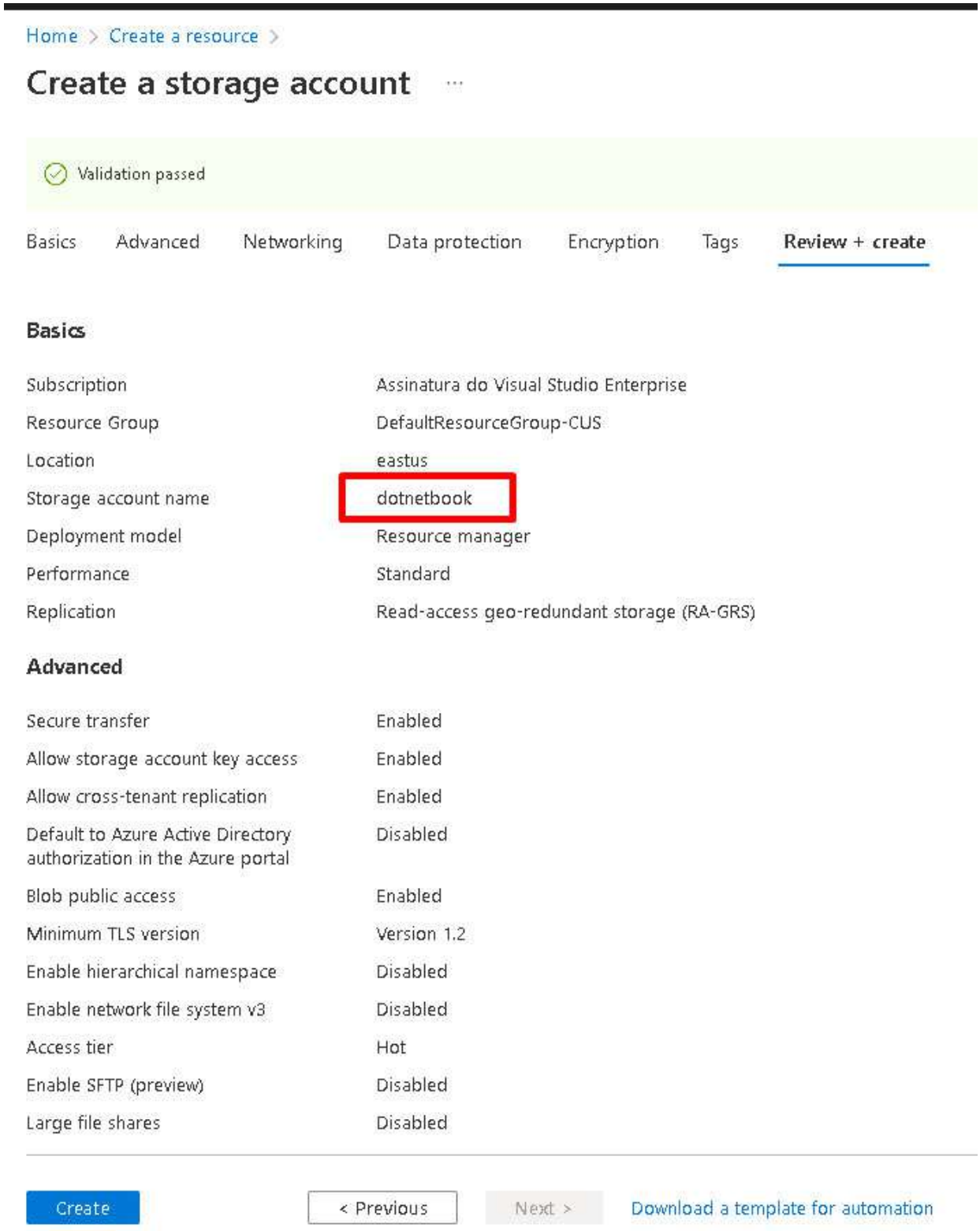


Figure 16.3: *Storage account creation*

If the name is unavailable on Azure, the creation will not be allowed, and a warning will be displayed in the underlying field. After confirming the creation, a progress status screen is displayed, as shown in *Figure 16.4*:

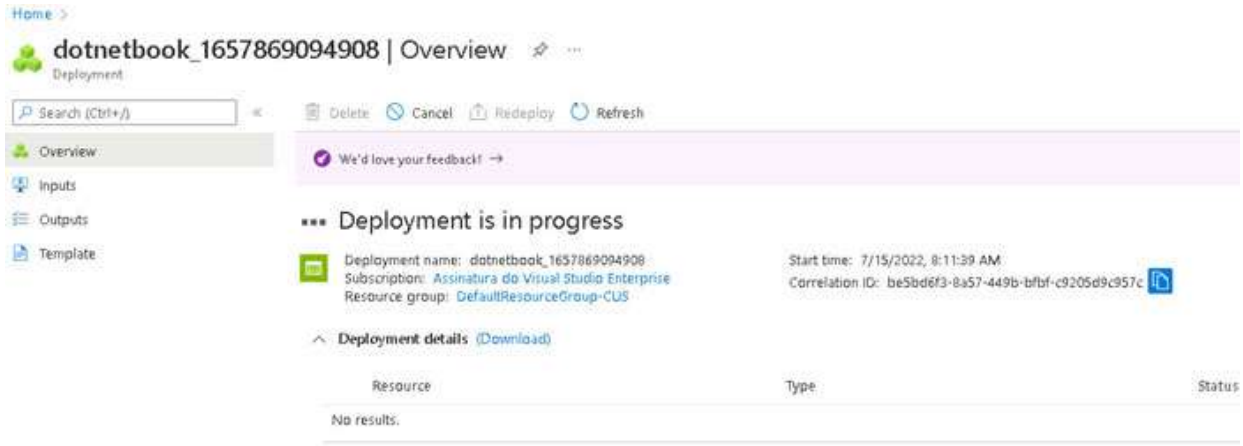


Figure 16.4: *Storage account creation status*

Once the deployment of the resource is complete, a correlation ID is provided, including more deployment details, such as file services, blob services, and much more, as shown in *Figure 16.5*:

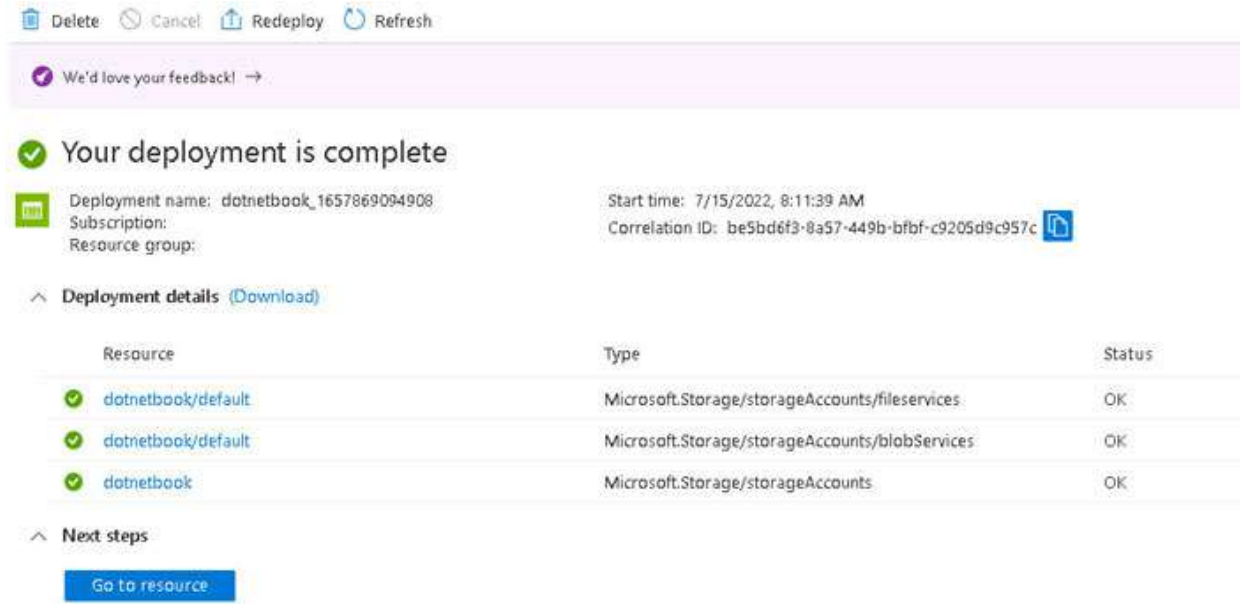


Figure 16.5: *Storage account deployment details*

After clicking on "**`Go to resource`**," you will be redirected to all the options available for your newly created storage account. As shown in *Figure 16.6*,

there are many services available within this type of resources, such as Containers, File Shares, Queues, Tables, and Azure CDN, including extra advanced options for Geo-replication, Data protection, Networking, Static Website, and much more:

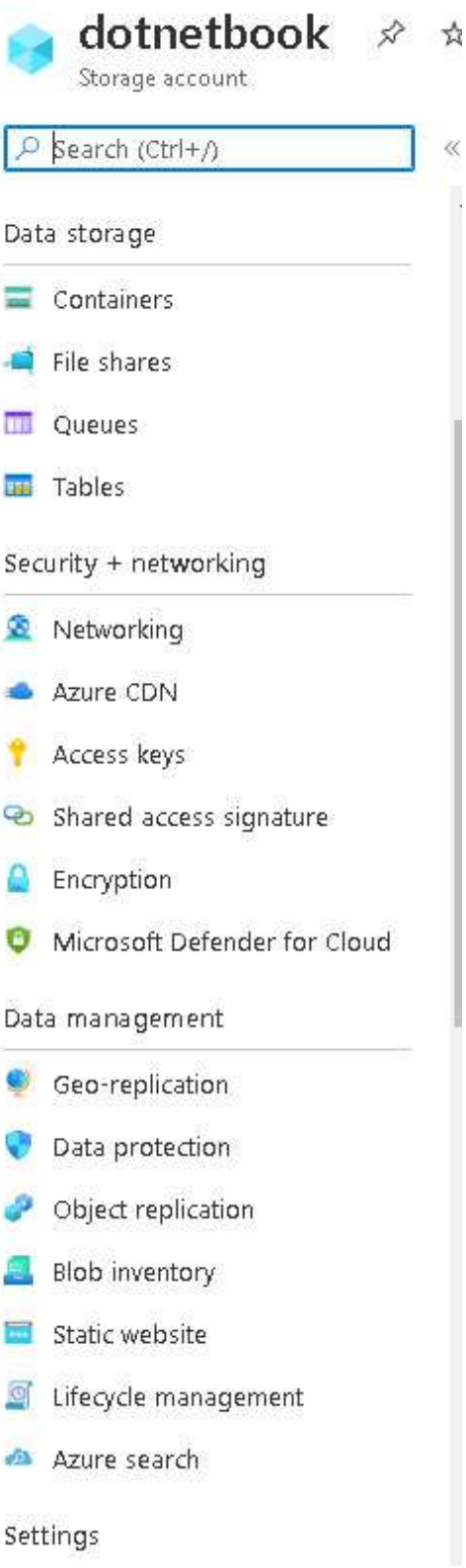


Figure 16.6: *Storage account options*

All the files within a Storage Account are stored within a container, where it is possible to specify access policies and to have a logical separation for all the files. To create your container, go to the underlying option and specify a unique name within your storage account for containers, as shown in *Figure 16.7*:

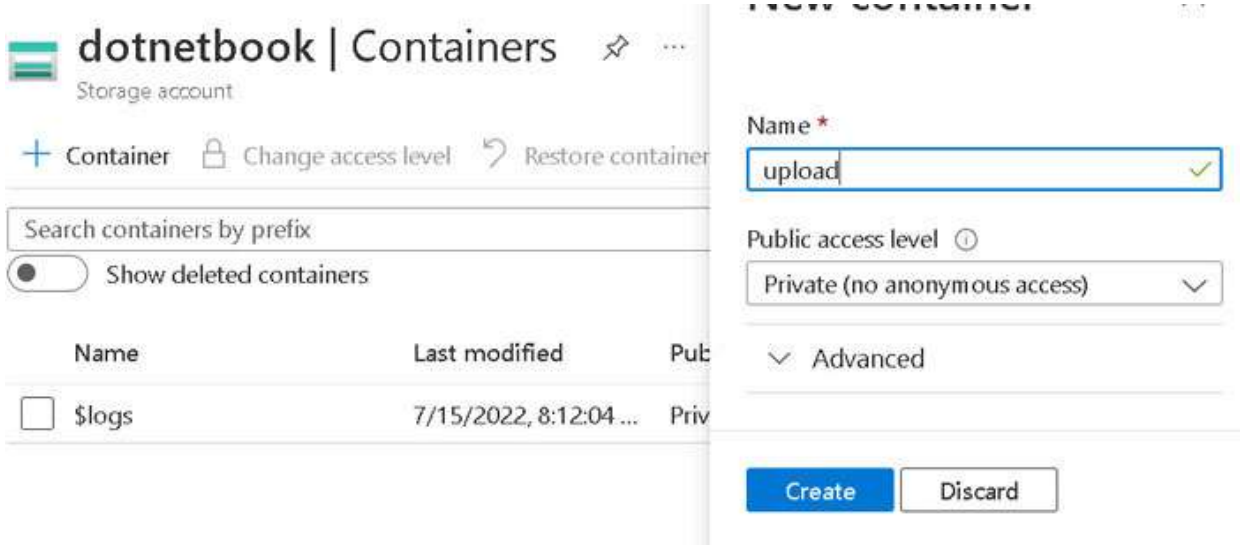


Figure 16.7: *Storage account container*

In this example, the name "**upload**" is specified for the container, and the Private access level is configured to require authentication to access any content within the container.

Considering you have the storage account already configured at this stage, the next step is to understand how you can integrate a .NET application with Storage accounts to upload, retrieve, update, and retrieve files. Within Visual Studio, create a new Console application and install the package **`Azure.Storage.Blobs`** provided by Microsoft, as shown in *Figure 16.8*:

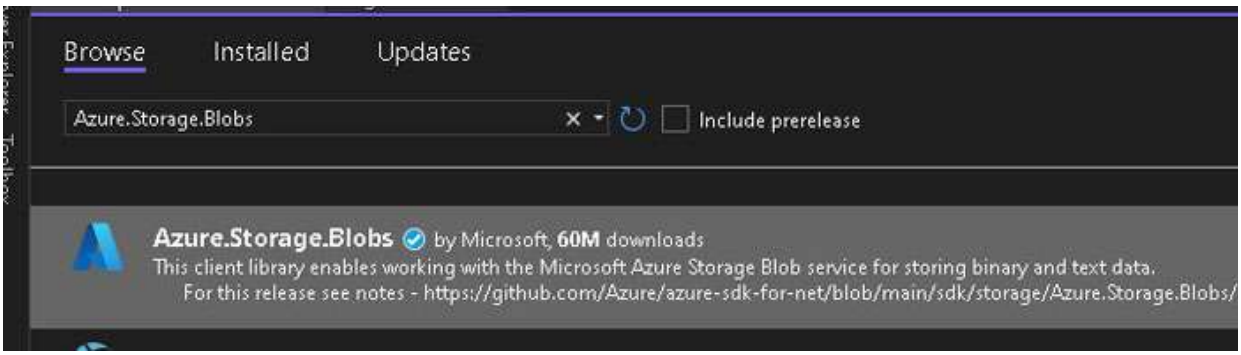


Figure 16.8: *Azure.Storage.Blogs package*

Within the Program.cs file, you would need to import a reference to the package and specify the authentication information regarding your storage accounts, such as **connectionstring** and container name, as shown in *Figure 16.9*:

```
Program.cs
BpbDotNetBook
1   using Azure.Storage.Blobs;
2
3   Console.WriteLine("Uploading file");
4
5   string azureAccountConnectionString = "YOUR CONNECTIONSTRING";
6   string containerName = "upload";
7   BlobContainerClient container = new BlobContainerClient(azureAccountConnectionString, containerName);
8   container.CreateIfNotExists();
9
10
11
12
```

Figure 16.9: *Authentication information for Azure Storage*

Note in *Figure 16.9* that the first code line contains the reference to **Azure.Storage.Blobs** package. Additionally, the fifth line contains a variable to store the connection string for the container, and line six has the container's name created under the underlying storage account. You can obtain the connectionstring for your storage account under the Access Keys option on Azure Portal, as shown in *Figure 16.10*:

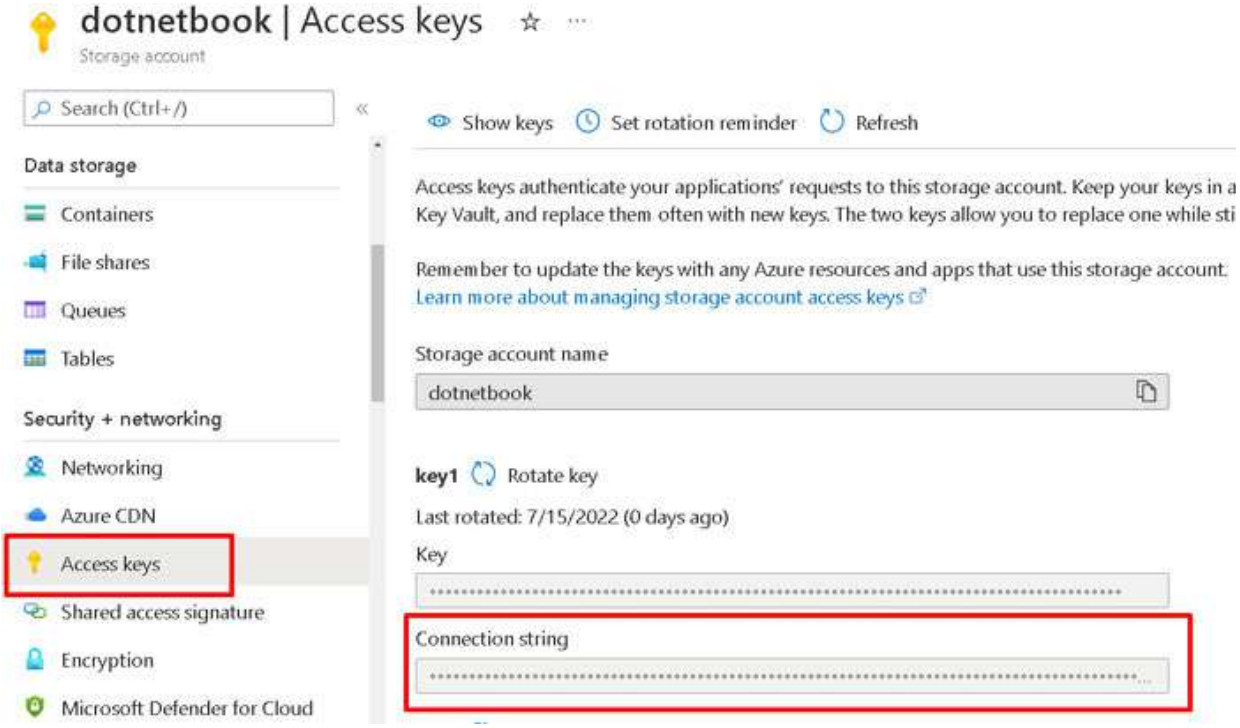


Figure 16.10: *Connectionstring for Azure Storage Account*

Once you get the **connectionstring,** you can replace the azureAccountConnectionString value. In real scenarios, this type of sensitive information should not be part of the codebase but a protected environment variable on the application server. Therefore, the hard-coded connectionstring is just for this sample code.

Looking at the continuation of the given code sample, note that lines 7 and 8 contain the creation of a BlobContainerClient instance, which is part of the package installed and allows us to establish communication with the Azure Storage Account passing the proper connectionstring and container information as highlighted in *Figure 16.11*:

```
BlobContainerClient container = new BlobContainerClient(
    azureAccountConnectionString,
    containerName);

container.CreateIfNotExists();
```

Figure 16.11: *Blob container client object*

In order to start uploading files to a specific container, it is essential to make sure the container gets created if that does not exist yet. Therefore, the method **`container.CreateIfNotExists()`** is called in this sample. In the context of this chapter, the container is called “upload.”

Once you have created the underlying container, the next step would be to upload an actual file to the container. To achieve that, you can create a file in your local machine, create a BlobClient object referencing the path that the file should be placed in the container, and finally call the Upload method, passing the file stream of your sample file, as shown in *Figure 16.12*:

```
string blobName = "sample.txt";
string filePath = $"myfiles/{blobName}" ;


BlobClient blob = container.GetBlobClient(filePath);

blob.Upload(File.OpenRead("C:\\Users\\Alexandre Malavasi\\Desktop\\sample.txt"));

Console.WriteLine("Upload done successfully");
```

Figure 16.12: *File upload to the Azure Storage container*

In the context of this chapter, a file called sample.txt was created manually and placed on the Desktop, and the underlying file path was passed as a parameter of the OpenRead method to get the underlying stream of the file. This code specifies that the sample.txt file will be uploaded into the upload container under the path "**`myfiles/sample.txt.`**" *Figure 16.13* contains the entire code for the example:

```
BpbDotNetBook
1   using Azure.Storage.Blobs;
2
3   Console.WriteLine("Uploading file");
4
5   string azureAccountConnectionString = "YOUR CONNECTIONSTRING";
6   string containerName = "upload";
7
8
9   BlobContainerClient container = new BlobContainerClient(
10      azureAccountConnectionString,
11      containerName);
12
13  container.CreateIfNotExists();
14
15
16  string blobName = "sample.txt";
17  string filePath = $"myfiles/{blobName}" ;
18
19
20  BlobClient blob = container.GetBlobClient(filePath);
21
22  blob.Upload(File.OpenRead("C:\\Users\\Alexandre Malavasi\\Desktop\\sample.txt"));
23
24  Console.WriteLine("Upload done successfully");
25
```

Figure 16.13: *Code sample for file upload*

After running the Console application within Visual Studio, the successful message regarding the upload will be displayed if all the parameters are correct, such as connectionstring, file path, and others, as shown in *Figure 16.14*:

```
Microsoft Visual Studio Debug Console
Uploading file
Upload done successfully

C:\Users\Alexandre Malavasi\source\repos\BpbDotNetBook\bin\Debug\net6.0\BpbDotNetBook.exe (process 5320) exited with cod
e 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure 16.14: *Successful file upload message*

After running the application, it is possible to access and see the underlying folder and file uploaded on Azure Portal, as shown in *Figure 16.15*:

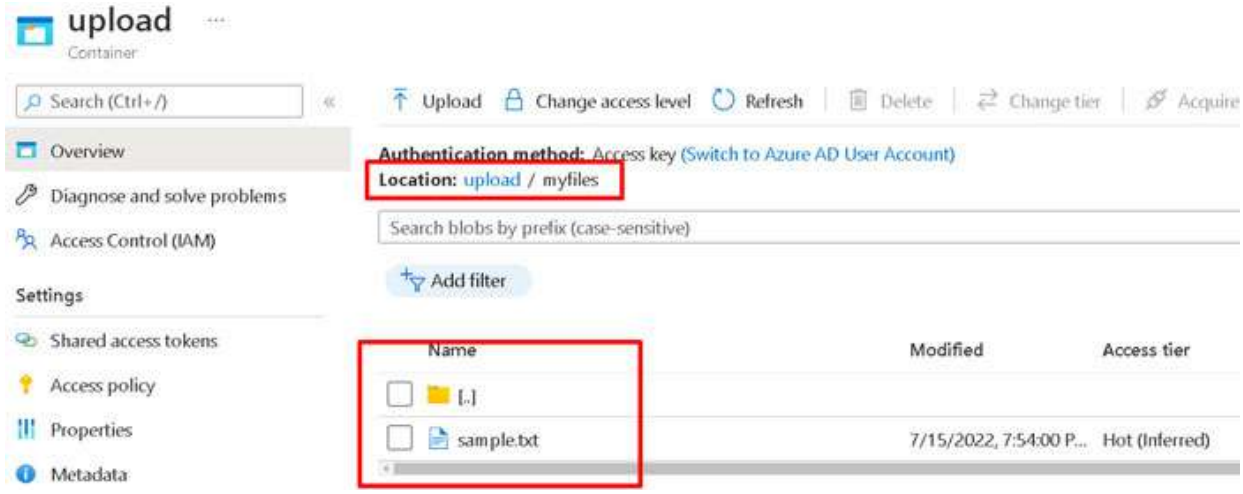


Figure 16.15: *Folder and file on the container*

The package to manipulate blobs in the storage account provided by Microsoft is quite powerful in terms of mirroring all the capabilities that are presented in this type of resource on Azure, allowing to upload, download, delete, get a list, change metadata, control security, and do any other operations for blobs in general.

Additionally, Storage Account has many other capabilities such as Tables, Queues, Networking configuration, Azure CDN, Static website, and much more, representing a great tool to extend the application's capabilities.

Azure functions

An Azure function is based on the serverless computing model that allows us to execute code using the events and triggers approach without having to take care of how the infrastructure for the host is configured, being a cost-effective option for scalable services.

Imagine you have a specific service that needs to be executed sporadically or only when a particular event happens. In this case, it does not make sense to keep a regular application server available 24 hours and seven days a week, increasing costs and overpricing the infrastructure for your service. It would be suitable to have a specific service that is online only when a particular event happens, and once the process is finished, the infrastructure would be turned off. Azure Functions are a perfect alternative for this operation, as it

allows us to create functions for specific purposes, billing the resource per use. Therefore, depending on the configuration of an Azure function app, it would allow millions of requests per month for a reasonable price.

To start using Azure Functions, a Function App on Azure itself must be created, which can host multiple functions together. If you go to Azure Portal within the Create a Resource option, you need to choose the Function App option, as shown in *Figure 16.16*:

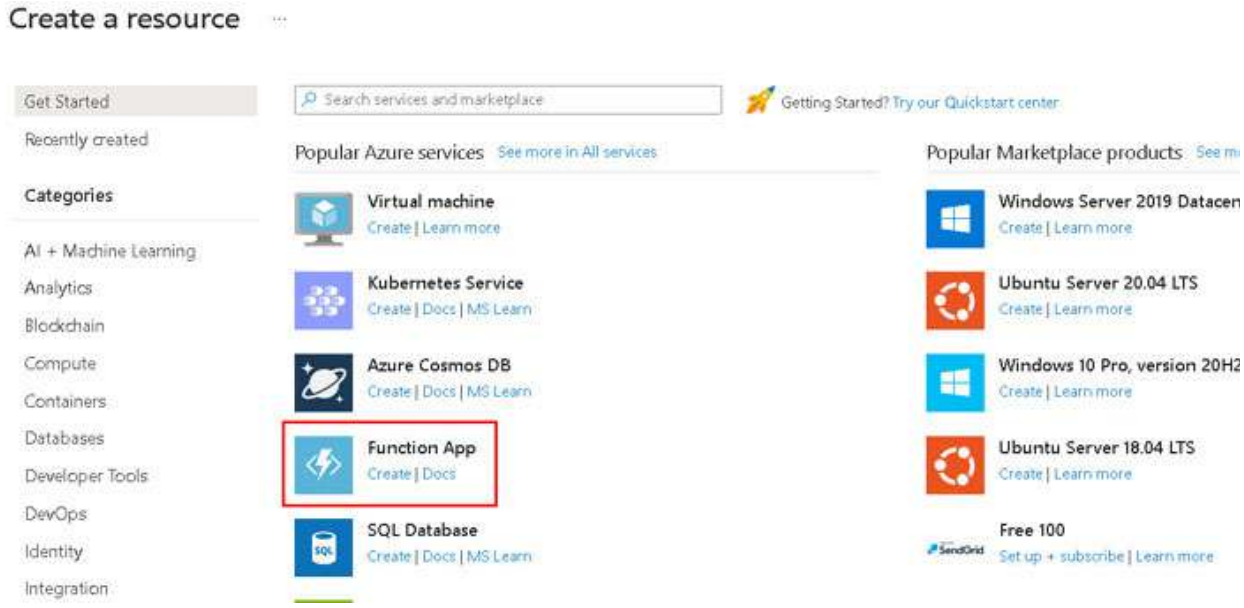


Figure 16.16: *Function App creation*

Similar to other resource types on Azure, it is mandatory to specify a unique name for your resource. It applies not only to your account but to all the resources presented on Azure as Function Apps use the domain azurewebites.net by default, and the name of your App cannot conflict with other existing Function Apps. You have to specify a subscription and resource group as well, as shown in *Figure 16.17*:

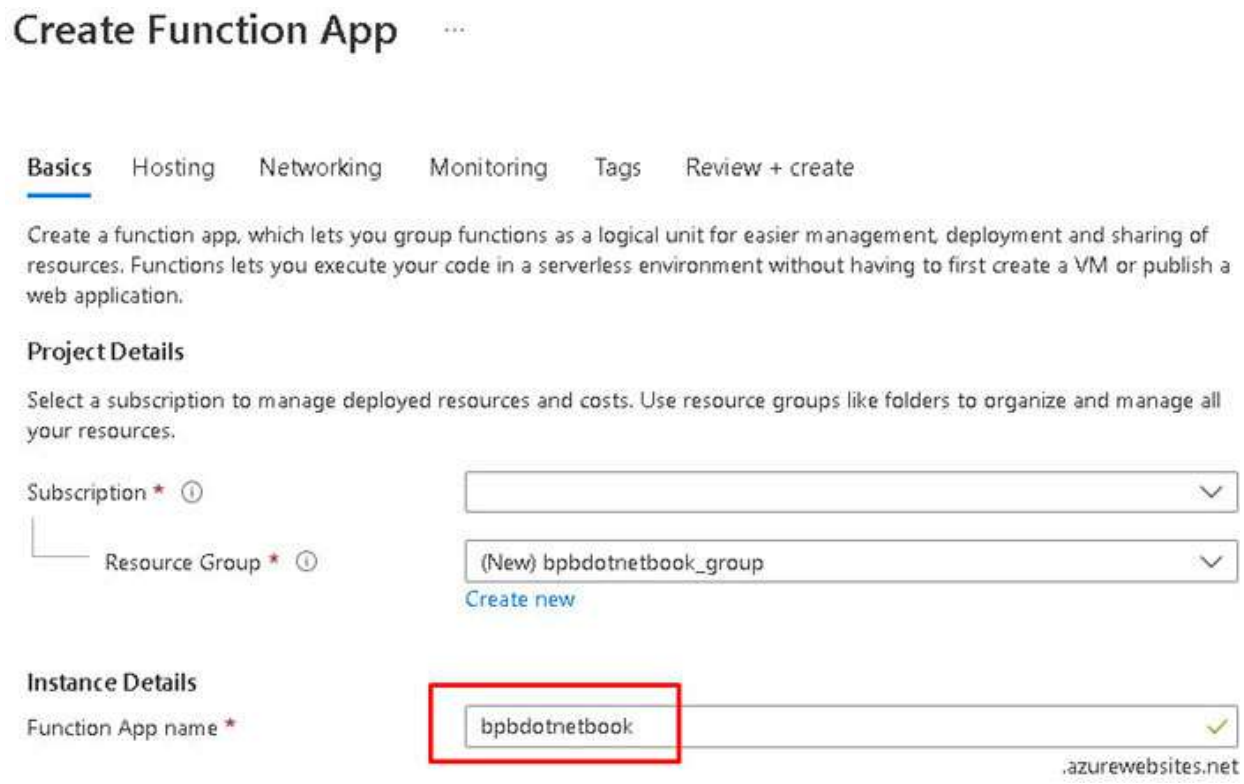


Figure 16.17: *Function app name*

Azure functions support code implementation using .NET, Node.js, Python, Java, and Powershell script. In the context of this chapter, it was chosen .NET as the primary stack development, with version six of the platform, as shown in *Figure 16.18*:

Figure 16.18: *Function app language*

The rest of the basic configuration regards Operating System and Plan, which directly influences the pricing model for keeping the app's infrastructure and executing the functions. In this example, the Linux operating system was chosen as .NET is multi-platform since the .NET Core

1.0 version, and the Consumption (Serverless) plan type was specified, as shown in *Figure 16.19*:

Figure 16.19: *Operating system and plan type*

After confirming the basic configuration, you are redirected to the preview screen, where you have the opportunity to check each relevant aspect of your app. The majority of configurations under the basic ones cannot be changed after their creation; therefore, it is essential to review the values specified in the last screen before confirming the actual creation, as illustrated in *Figure 16.20*:

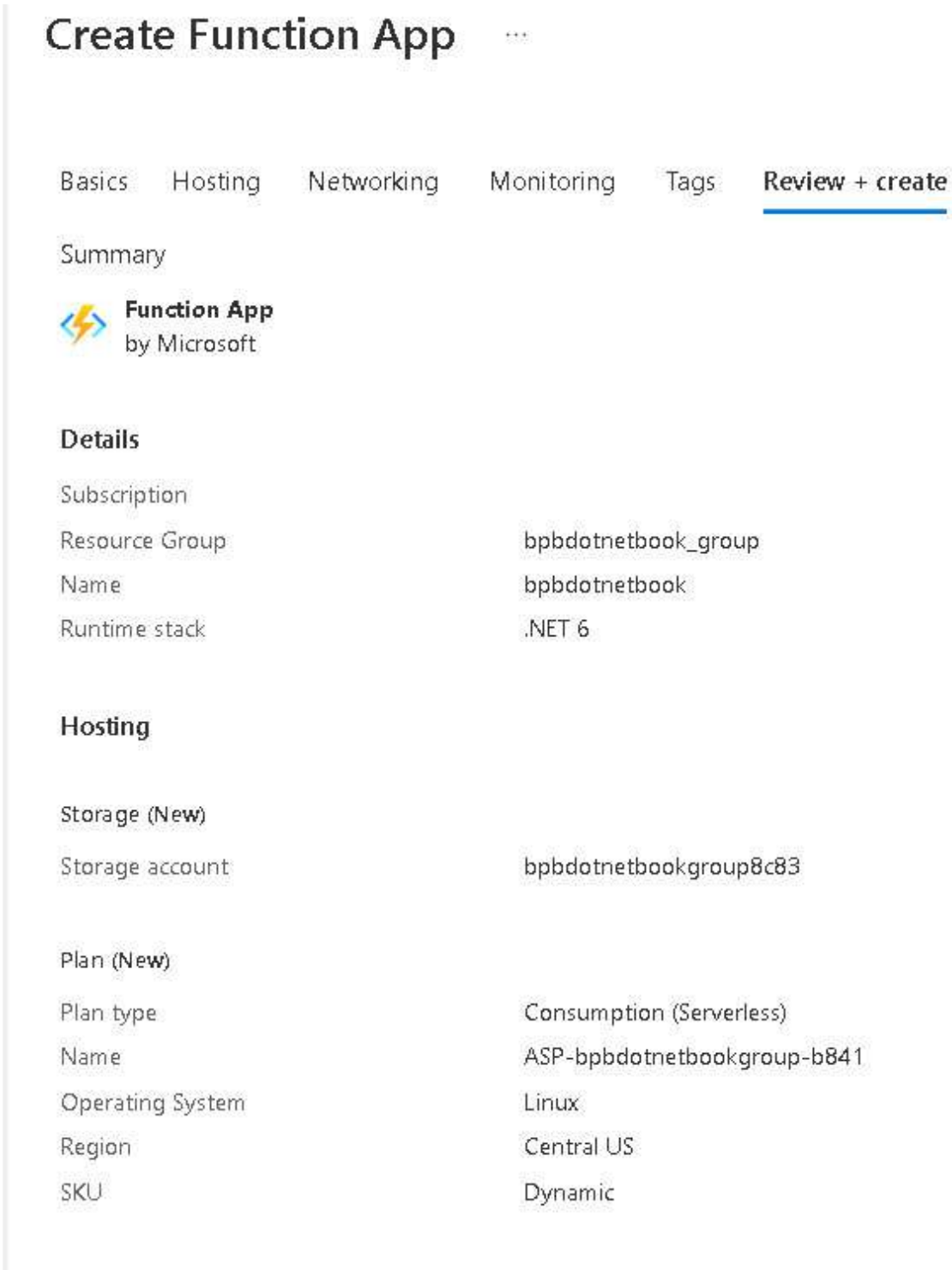


Figure 16.20: *Function app review screen*

If the deployment succeeds, you should be able to see the details of all the resources that were created, including a storage account, sites, and components, as shown in *Figure 16.21*:

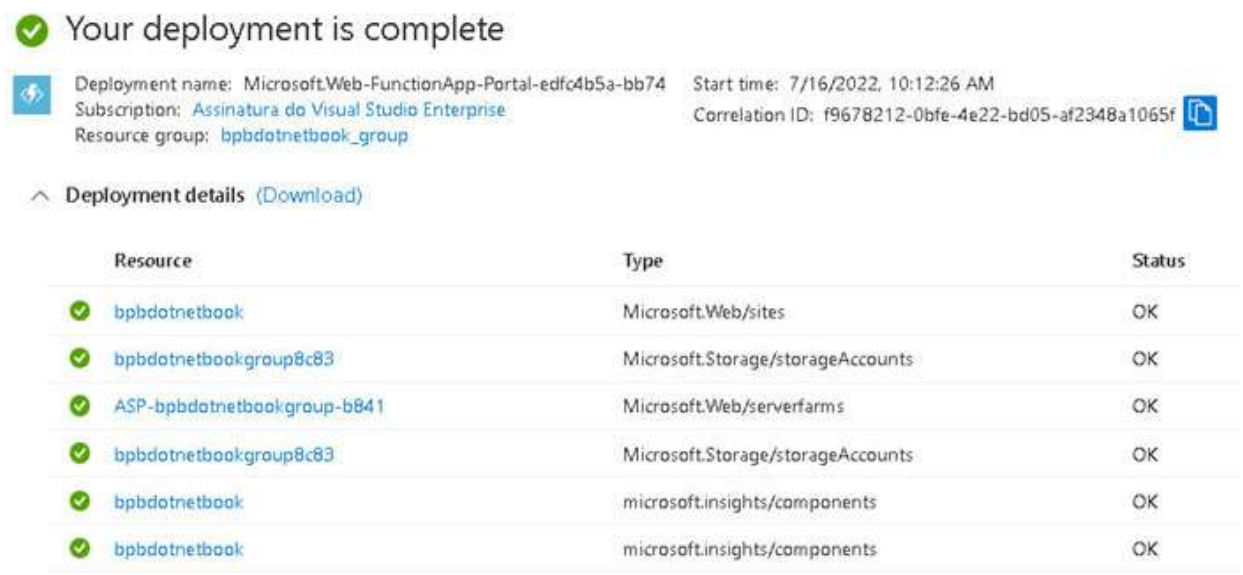


Figure 16.21: *Function app deployment details*

After the Function App is created, you can manually create functions, as shown in *Figure 16.22*:

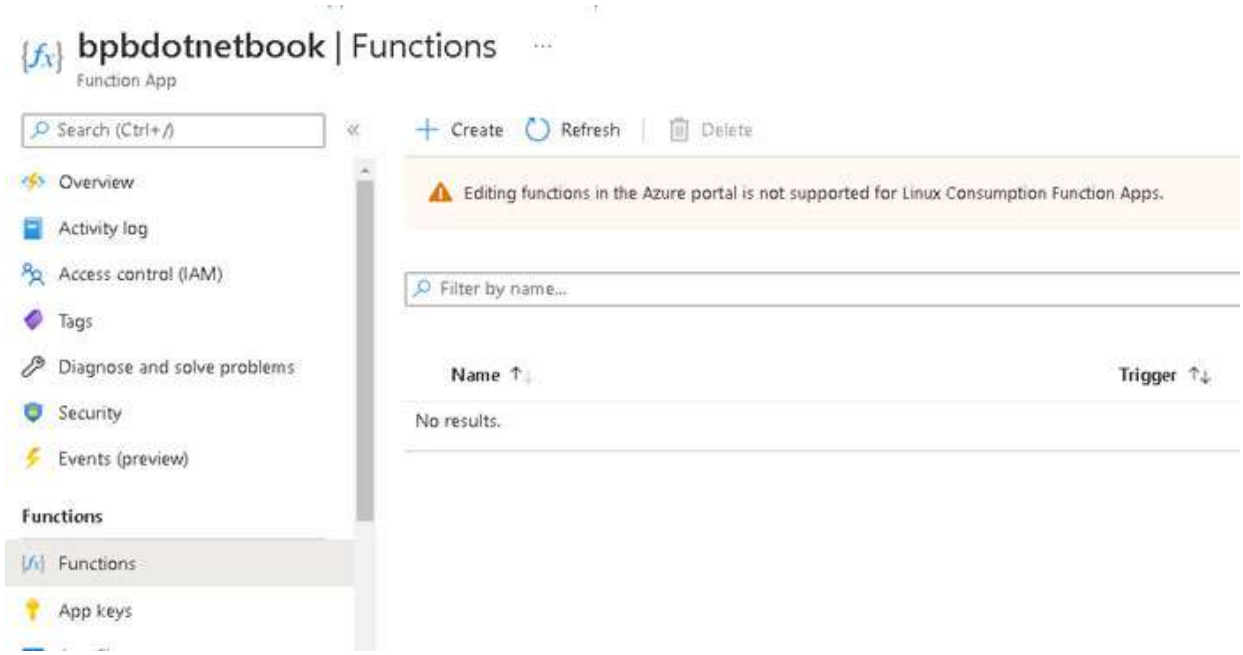


Figure 16.22: *Functions on Azure Portal*

As Azure functions host a piece of code to be executed, it is possible to develop the functions locally using Visual Studio or Visual Studio Code. In this case, the underlying Azure development workload in Visual Studio needs to be installed, as shown in *Figure 16.23*:

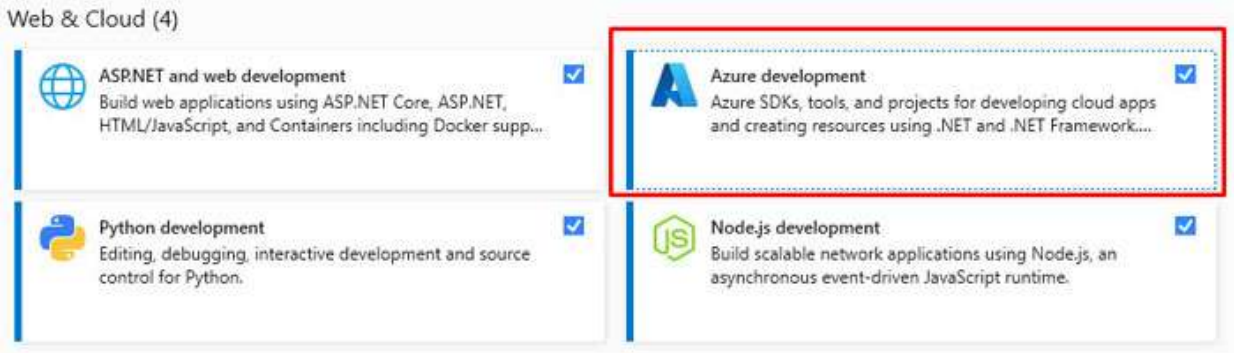


Figure 16.23: *Azure development workload*

After the installation of the workload, all the relevant project templates for Azure development are available in Visual Studio when you start a new project from scratch. To experience local development for Azure Functions, you would need to create the underlying project in Visual Studio, as shown in *Figure 16.24*:

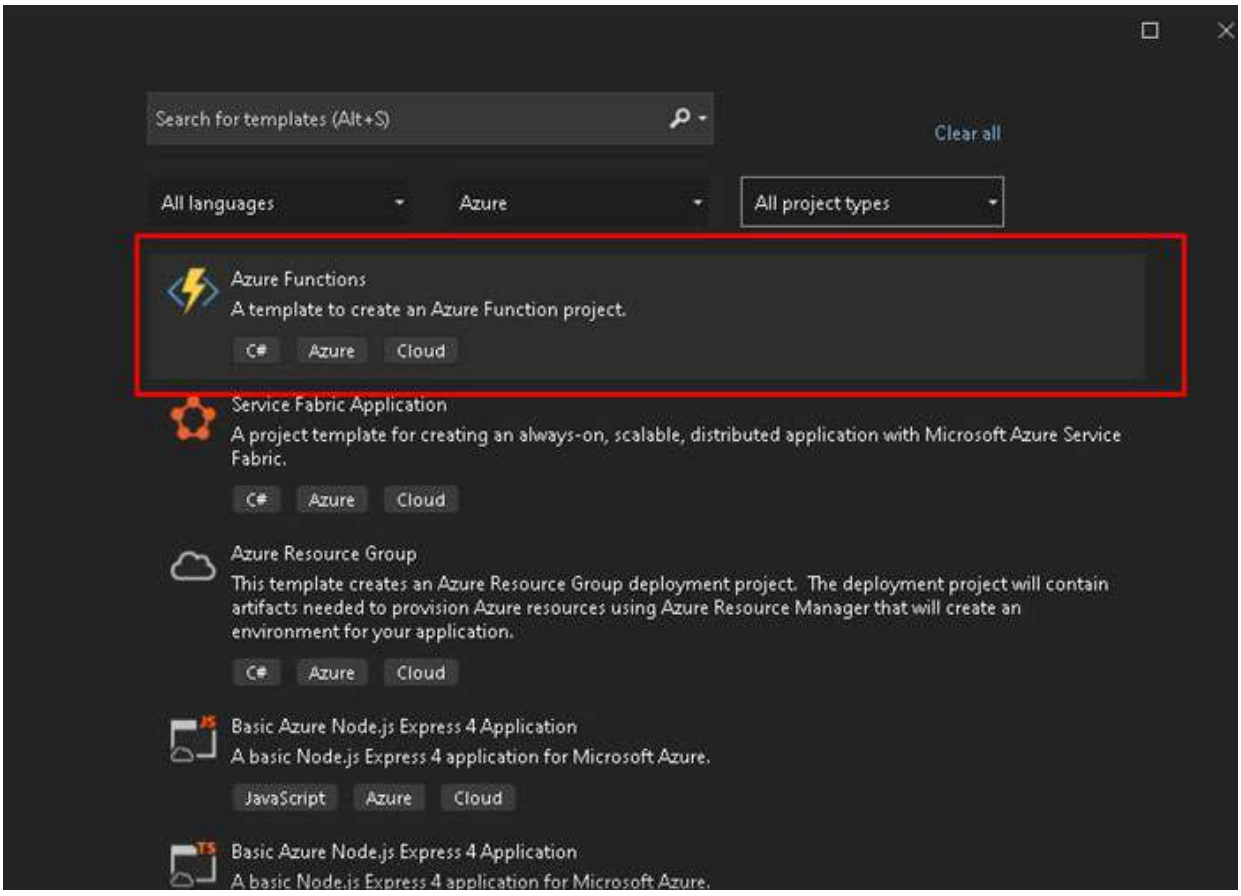


Figure 16.24: *Azure function project in Visual Studio*

After choosing the underlying project type and giving the project a name, you must choose the .NET version, the trigger type, and the authorization level. In the samples of this chapter, the .NET 6 version was chosen, an HTTP trigger and Anonymous access, as shown in *Figure 16.25*:

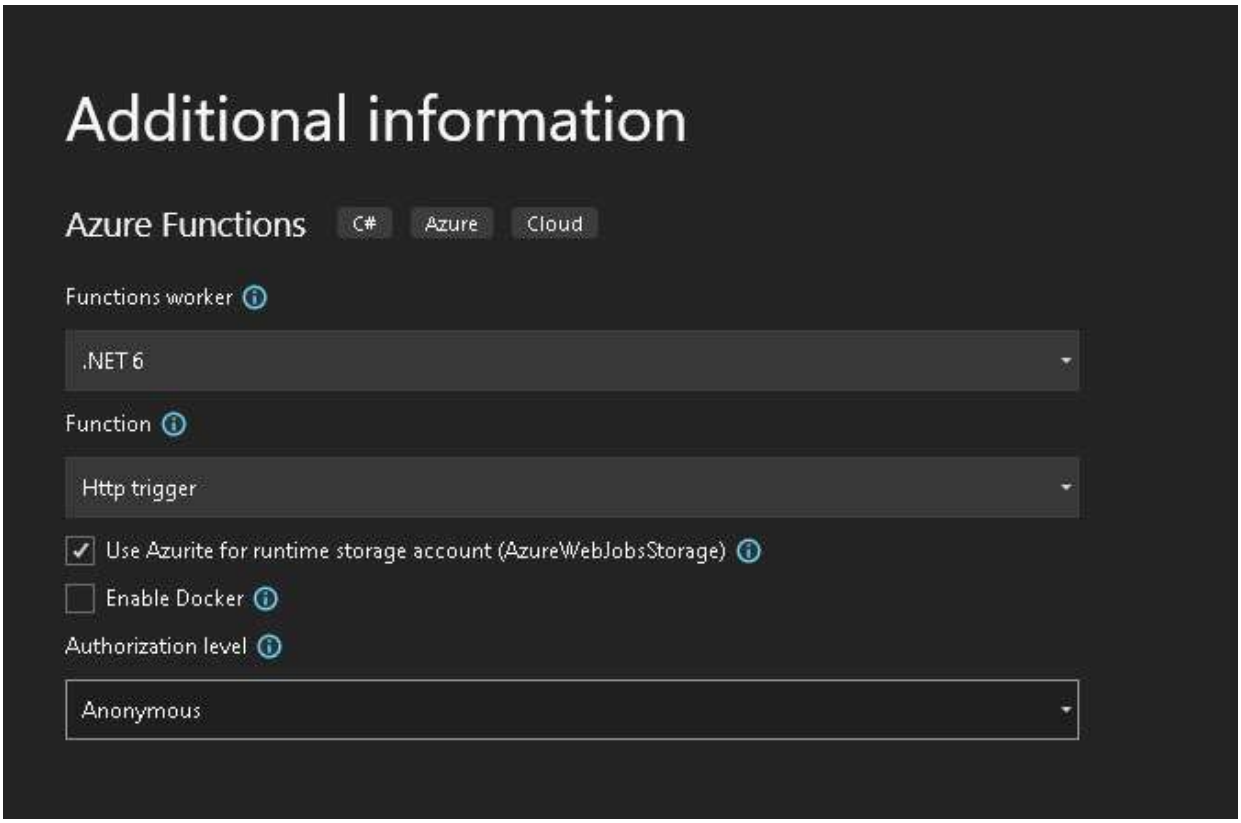


Figure 16.25: *Azure function project configuration*

The HTTP trigger option means that the function will be executed once the Web endpoint for the function is called: GET, POST, or PUT, depending on the configuration. Regarding the authorization level, in production environments, another type of authentication should be used for security reasons, but for testing and studying purposes, Anonymous access is used in this chapter's context.

The default project template for the HTTP trigger type creates a static class called Function1, a local settings file, and information on the host, as shown in *Figure 16.26*:

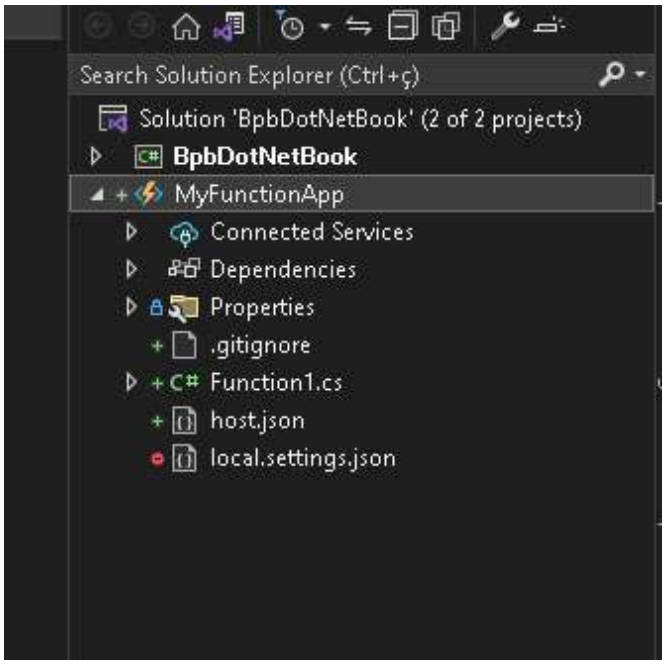


Figure 16.26: *Solution Explorer for Function Apps*

If you click on the Function1 file, you can see the default function created as an example by the template. It contains the configuration for the authorization level, allowed HTTP Verbs, and route configuration. Considering the context of any application based on HTTP requests and endpoints, it is possible to say that a function app based on an HTTP trigger is similar to a standard RESTful Web API, with a combination of the following:

- Authentication and authorization configuration
- Multiple methods
- Multiple types of action results
- HTTP verb and route configuration
- Dependency injection

Therefore, if you are familiar with Asp.Net Core Web API projects, understanding Azure Functions will be facilitated for specific trigger types. After creating a project with the default project template, the Function1 class looks like this in *Figure 16.27*:

```
11  namespace MyFunctionApp
12  {
        0 references
13      public static class Function1
14      {
15          [FunctionName("Function1")]
            0 references
16          public static async Task<IActionResult> Run(
17              [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpR
18              ILogger log)
19          {
20              log.LogInformation("Developing my first Azure Function.");
21
22              string name = req.Query["name"];
23
24              string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
25              dynamic data = JsonConvert.DeserializeObject(requestBody);
26              name = name ?? data?.name;
27
28              string responseMessage = string.IsNullOrEmpty(name)
29                  ? "This HTTP triggered function executed successfully. Pass a name in the
30                  : $"Hello, {name}. This HTTP triggered function executed successfully.";
31
32              return new OkObjectResult(responseMessage);
33          }
34      }
35  }
```

Figure 16.27: *Function1 code*

GET and POST HTTP verbs are allowed in the given function, and the authorization is marked as Anonymous on lines 16 and 17. The method gets the value of a "**`name`**" parameter passed as part of the URL request and returns a string response with the message: "**`Hello. This HTTP triggered function executed successfully`**". If you run the application using Visual Studio, a simulated Azure Function App infrastructure will be executed locally, and a localhost endpoint will be provided, as shown in *Figure 16.28*:

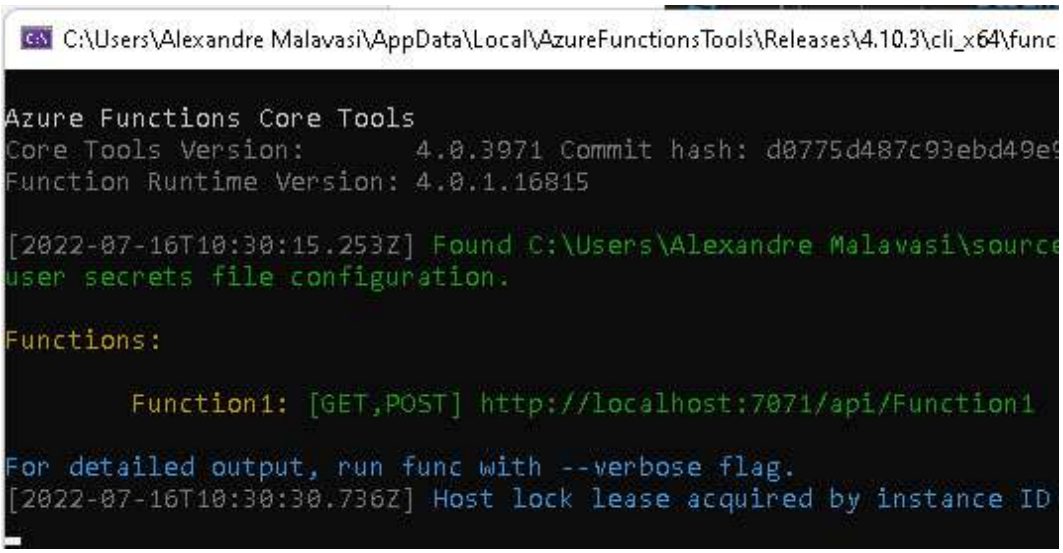


Figure 16.28: *Azure function local execution*

The endpoint **http://localhost:7071/api/Function1** was created, and it can be executed directly in the browser as a standard Web API, as shown in *Figure 16.29*:

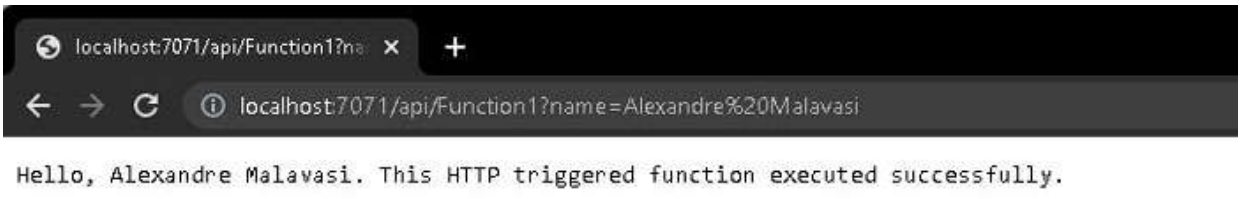


Figure 16.29: *Azure function browser execution*

Please note in the URL bar that the name parameters were passed as part of the request with “Alexandre Malavasi.” Considering the configuration of the function code for the Function1 method, the correct output is displayed, as highlighted in *Figure 16.30*:

```
0 references
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)] HttpR
    ILogger log)
{
    log.LogInformation("Developing my first Azure Function.");

    string name = req.Query["name"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    string responseMessage = string.IsNullOrEmpty(name)
        ? "This HTTP triggered function executed successfully. Pass a name in the
        : $"Hello, {name}. This HTTP triggered function executed successfully.";

    return new OkObjectResult(responseMessage);
}
}
```

Figure 16.30: *Azure function output*

To test the execution of the azure function developed locally, it is possible to deploy the same code to the Azure Function App created previously in this chapter. To achieve that for studying purposes, you can use the option "**`Publish Selection`**" under the "**`Build`**" menu option in Visual Studio, as demonstrated in *Figure 16.31*:

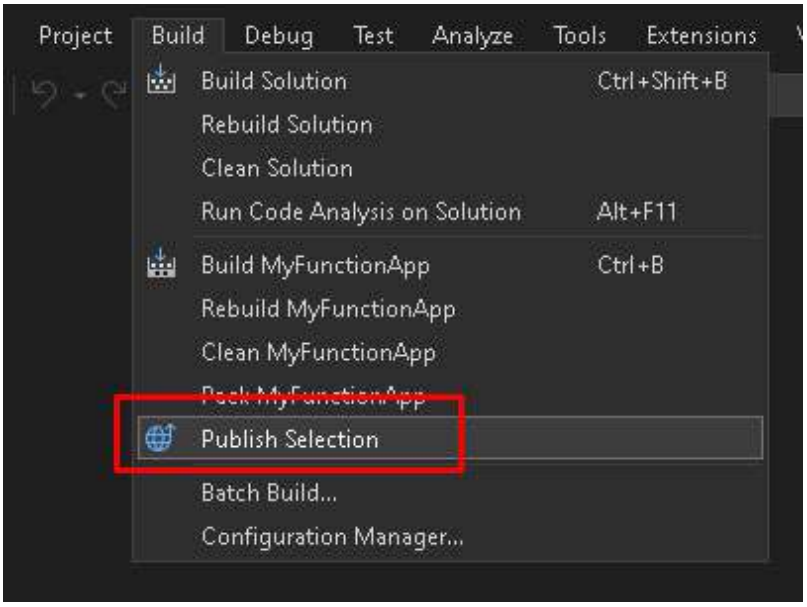


Figure 16.31: *Publish selection*

After choosing this option, four options are displayed, including the Azure deployment option, which is the correct option in the context of this chapter,

as shown in *Figure 16.32*:

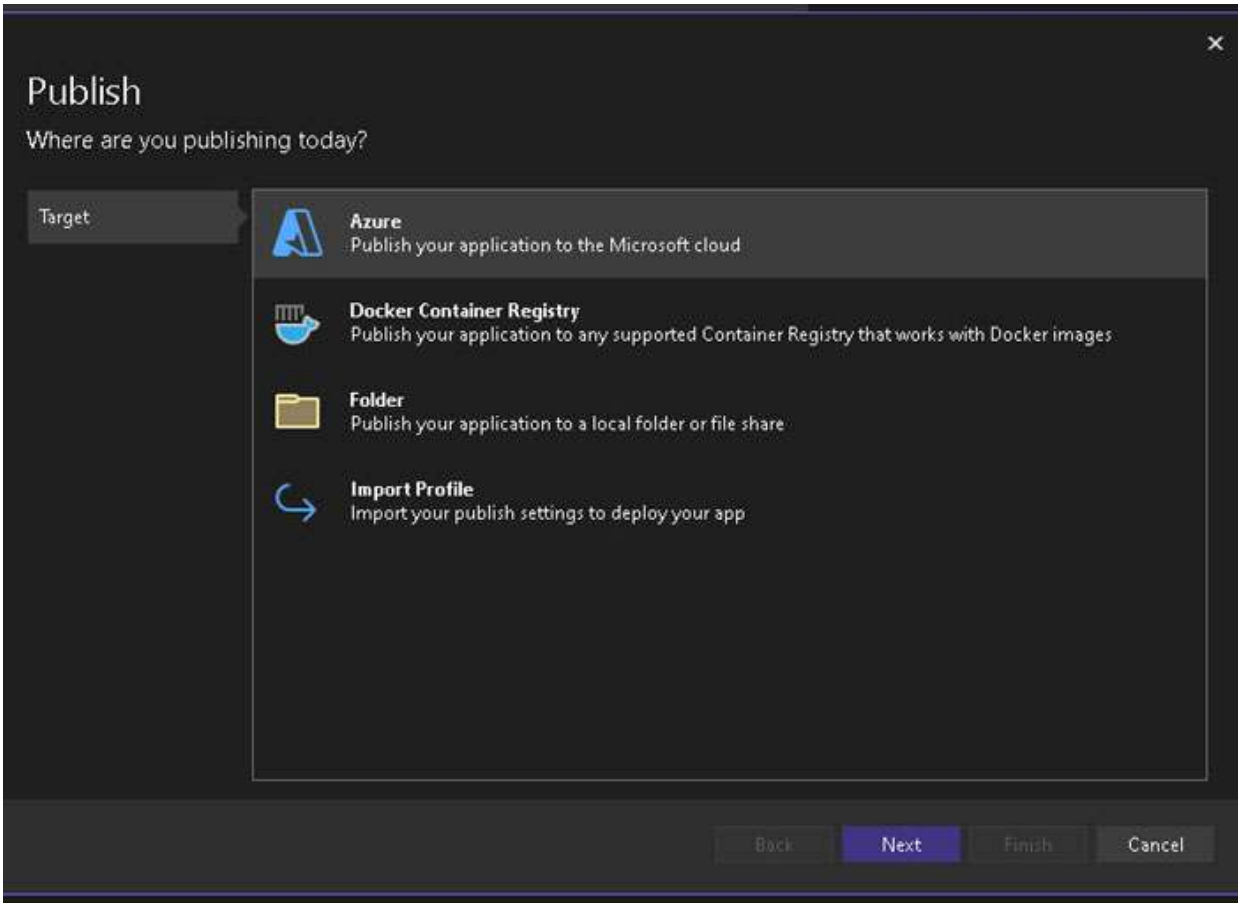


Figure 16.32: *Azure publish option*

Azure offers the deployment of Azure functions for Windows and Linux. Furthermore, it is possible to deploy the function app using Container. As the Function App created previously in this chapter is based on Linux operating system, this underlying option needs to be chosen, as shown in *Figure 16.33*:

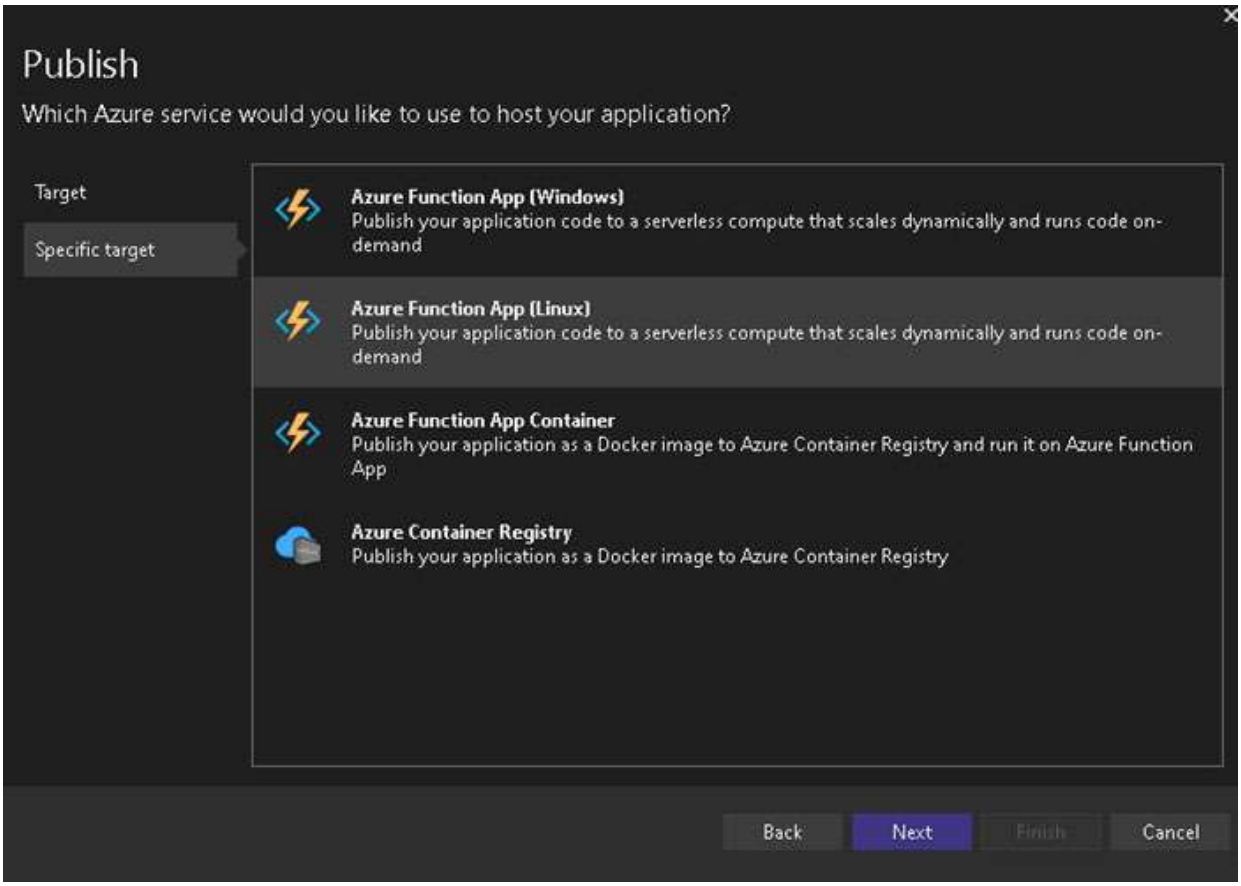


Figure 16.33: *Azure function deployment in Linux*

Considering the deployment needs to happen in an existing Azure subscription under a Microsoft account if would need to authenticate with your own Microsoft Account and select the underlying subscription under your account. After this, all the existing Function Apps based on Linux are displayed, as shown in *Figure 16.34*:

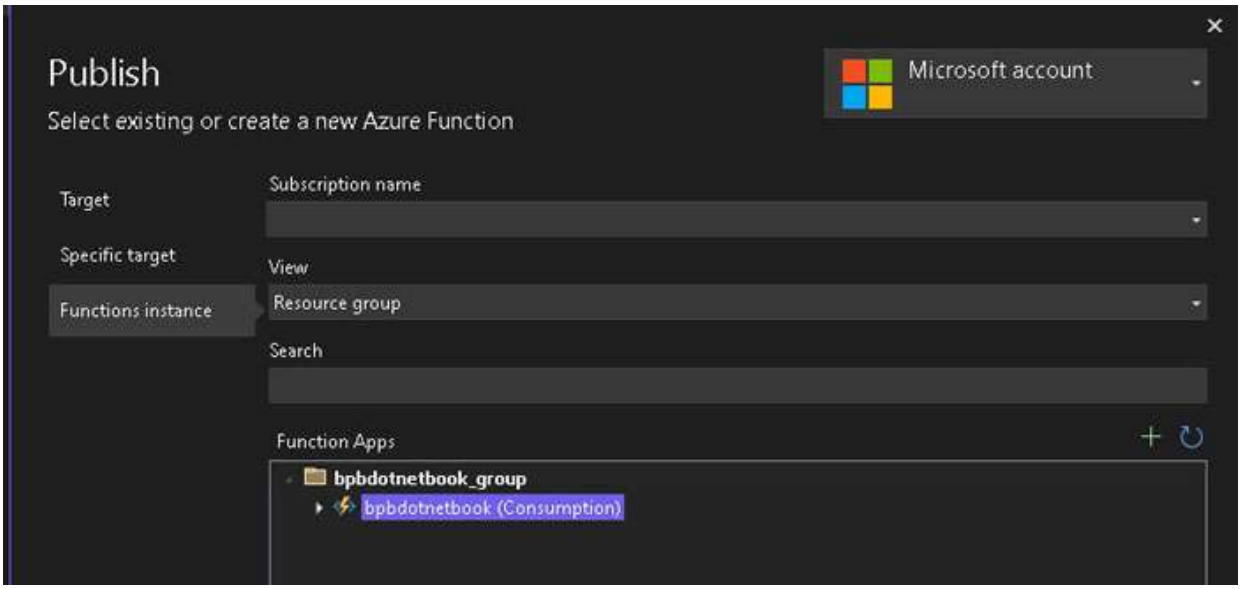


Figure 16.34: *Azure Function Apps based on Linux*

After finishing the configuration for the publication process, the underlying profile for the publication process is created, which can be reused to automate the publishing process by exporting the related file. The publication profile should look like *Figure 16.35*:

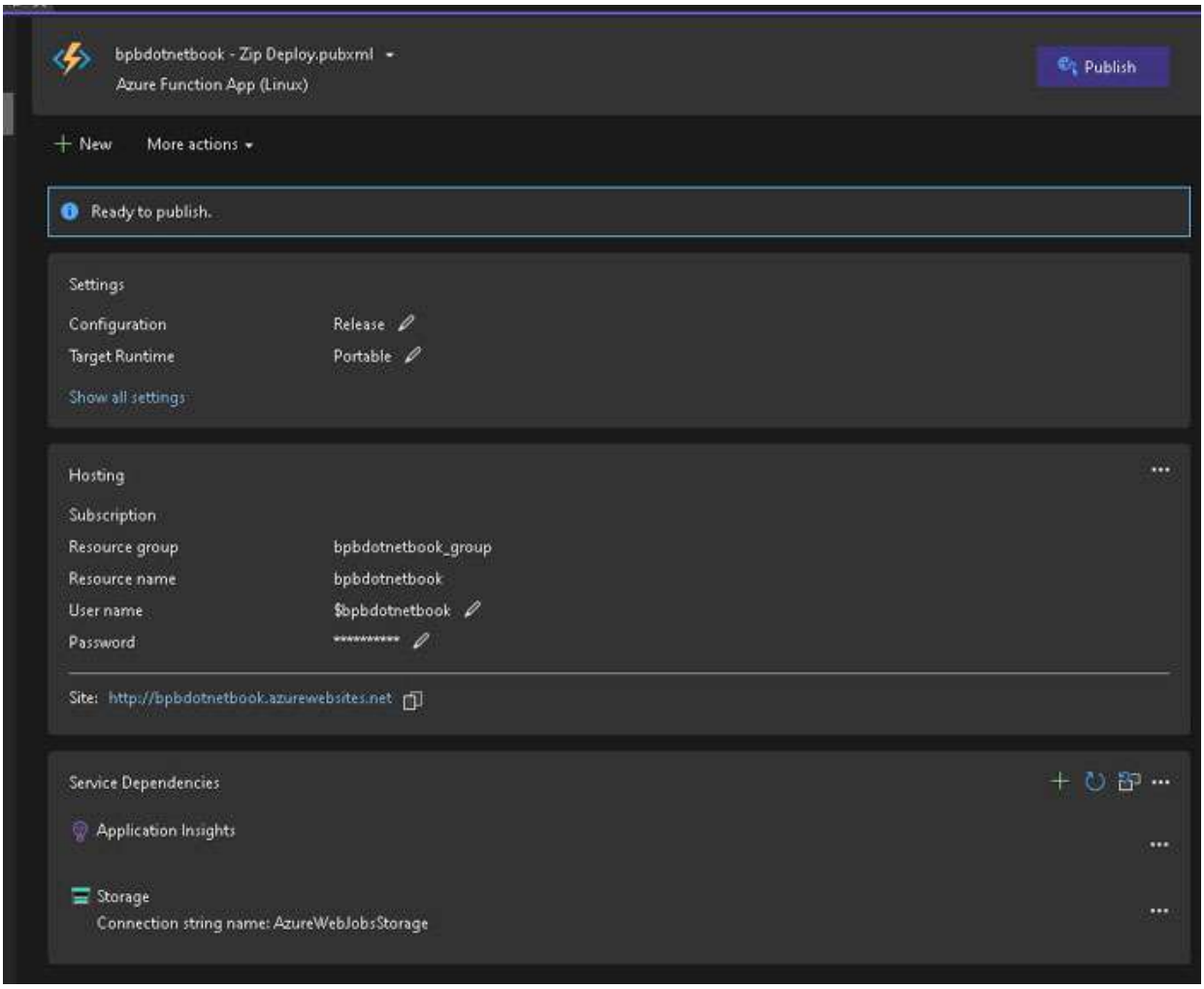


Figure 16.35: *Publication profile*

After pressing "**`Publish`**," the package for your Azure Function app will be pushed to the infrastructure created on Azure, and you should be able to execute your function using the Azure sub-domain for your function app. In the context of the one created in this chapter, the URL would be **https://bpbdotnetbook.azurewebsites.net**.

Conclusion

In this chapter, you had the opportunity to learn the most basic possibilities of integration with Azure services using the C# language and .NET project types, such as Azure Functions and Blob Storage services.

Azure contains a vast amount of services that can be consumed via standard REST APIs; however, there are libraries and SDKs available for the most

used services within Azure, available not only for C# but for other languages as well, such as Node and Python, which facilitates and speeds up the development process when the solution involves Cloud services at any level.

In the next chapter, you will learn how to include authentication and authorization for .NET applications focused on Web development.

Points to remember

- It is recommended to place the connectionstring for your Azure resources into proper and secure secret settings or environment variables.
- Azure Functions support Python, Java, Node.js, .NET, and PowerShell.
- It is possible to host Azure Function Apps in Linux or Windows, depending on the region selected.
- Azure SDK for Blob Storage allows us to upload, download, update, and delete files on Azure Storage Accounts, including manipulating metadata.

Multiple-choice questions

1. **Which alternative represents a false statement for Blob Storage configuration?**

    a. There is a restriction on uploading PDF files in containers.

    b. All the containers are public by default.

    c. The URL must be unique.

    d. It is possible to configure access control per container.

2. **Which .NET versions are supported for Azure Functions?**

    a. .NET Framework 2.0, .NET Core 1.0, .NET Core 3.1, and .NET 5

    b. All .NET Core versions

    c. .NET 5 and .NET 6

    d. .NET Core 3.1 and .NET 6

3. **Which service is not supported by Azure Storage Accounts?**

a. Tables
b. File Share
c. Static website
d. SignalR

Answers

1. **b**
2. **d**
3. **d**

Questions

1. Explain what is a serverless application.
2. Create an Azure function that accepts only POST HTTP Verb and returns a querystring parameter.
3. Create a Console App that uploads a file into an Azure Blob Storage container and downloads the same file into a local folder.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 17

Authentication in Asp.Net Core

Introduction

Security is one of the main aspects of any Web application these days, as data confidentiality and trust are the key elements for any successful business which has software development at its core. Asp.Net Core contains native mechanisms for authentication and authorization for Blazor, Web APIs, and MVC applications, using a standard model across different projects.

Learning authentication and authorization for Asp.Net Core applications will allow you to build trustable and reliable Web applications, speeding up the development process in terms of productivity, as any enterprise application needs a certain level of security and compliance with market standards for data confidentiality.

This chapter will cover the fundamental aspects of authentication for the most common Asp.Net Core project types, including Blazor, Web API, and MVC, including good practices in terms of the authorization.

Structure

In this chapter, we will discuss authentication and authorization for Asp.Net Core applications in the following topics:

- Authentication concepts
- Authentication and authorization for Web APIs
    - Basic authentication
    - JWT authentication

Objectives

After studying this unit, you will understand the basic concepts of authentication and authorization for Asp.Net Core applications. You will learn how to apply different types of authentication and use basic authorization concepts for Asp.Net Core applications.

Authentication concepts

Authentication and authorization are essential aspects of any enterprise application as it protects user data and gives the application reliability. Authentication is the process responsible for identifying the user's identity. Basically, for determining if users requesting authentication to an application are who they are saying they are. Usually, an authentication process involves informing a username and a password. If the username and password given to the application match with the username and password provided by a particular user, it is enough to determine the user’s identity, as there is an assumption that the user is the only one with ownership of the informed credentials. Authentication can also involve a more complex process, such as verifying the user's IP, network, geo-localization, token, and anything else that helps secure the user’s identity.

Enterprise applications can use many types of user authentication, such as passwords, security questions, multi-factor authentication, tokens, mobile phones, and **One Time Pin** (**OTP**) via SMS or biometric authentication. This chapter uses a username and password to determine the user's identity.

Even though authentication and authorization are used in the same context, they are different things. Authentication only determines the user’s identity, and authorization determines the user's access level. In terms of authorization, imagine a configuration that would restrict users per profile or role, determining which pages and actions on these pages each user should have access to and perform. Based on the user identity provided by the authentication process, an application can verify what actions a specific user can make while navigating through a Web application.

Authentication and authorization for Web APIs

The .NET platform follows a similar pattern for implementing authentication and authorization among all the Web project types available, including Web API, MVC, Razor, and Blazor projects. In this section, we will walk you

through how the implementation is done for Web APIs in general, with some code samples following a step-by-step approach.

Considering any Web API needs to be hosted in a server infrastructure to be available for other consumers, the authentication process is highly tied to the intrinsic capabilities of the server. The HTTP modules are used for authentication if the application is hosted in a traditional **Internet Information Service** (**IIS**) infrastructure. With the possibility to extend or overwrite IIS modules according to your convenience if the application requires a custom authentication.

In a traditional Web API, the host manages the authentication process's state, returning an object to the application in the context of a Web request. This object contains the identity information and is attached to the current thread in the request context. This object implements the **IPrincipal** interface, which is pretty much standard across .NET projects.

Suppose your application is not hosted in IIS. In that case, it is possible to achieve similar authentication goals using **HTTP Message Handlers**, which allows you to set the **principal** object by yourself. This authentication scheme is exciting where a more custom authentication process needs to happen or when the application requires to be host-agnostic.

Usually, an authentication process involves matching the username and password with values stored in the database or even another more sophisticated approach. To simplify the examples in this chapter, they will not contain details on how the user is retrieved from a database, as each developer or project may retrieve a user from different locations or in particular ways. Therefore, this section restricts to the necessary code to set the principal user, as shown in *Figure 17.1*:

```
private void SetPrincipal(IPrincipal principal)
{
    Thread.CurrentPrincipal = principal;
    if (HttpContext.Request != null)
    {
        GenericIdentity userIdentity = new GenericIdentity(principal.Identity.Name);
        GenericPrincipal userPrincipal = new GenericPrincipal(userIdentity, null);

        HttpContext.User = userPrincipal;
    }
}
```

Figure 17.1: *Set Principal object*

If you are hosting the application in IIS, you must set the principal in the **`Thread.CurrentPrincipal`** and **`HttpContext.User`** properties, as the HTTP context is available for an application hosted with HTTP Modules. But, if you are hosting the Web application another way, you must set only the **`Thread.CurrentPrincipal`** property.

In terms of authorization for Web APIs, as all the methods are based on Controllers within a request in Asp.Net Core Web API, a validation in terms of authorization is done before the request reaches controllers. This process happens using **Authorization Filters**, which represents good practice for decoupling the implementation of the filter from the actual controller method implementation.

Asp.Net Core has its own Authorization Filter, which returns the 401 code status if the user is not authenticated and/or does not have the authorization to access the underlying resource. You can configure the authorization filter by specifying it on each of the following levels:

- Action Controller
- Controller
- Globally

In a controller, you can become the entire class with the required protection in terms of authentication and authorization by using the Authorize attribute, as highlighted in *Figure 17.2*:

```
3   using Microsoft.AspNetCore.Authorization;
4
5   namespace BPBWebApi.Controllers
6   {
7       [ApiController]
8       [Authorize]
9       [Route("[controller]")]
            3 references
10      public class CustomerController : ControllerBase
11      {
12          private readonly ILogger<CustomerController> _logger;
13
                0 references
14          public CustomerController(ILogger<CustomerController> logger)...
18
19          [HttpGet]
                0 references
20          public Customer GetCustomer(int id)...
```

Figure 17.2: *Set Authorize attribute for controllers*

If the **`authorize`** attribute is used at a controller level, it means that all the methods within this controller inherit the same level of authorization and authorization filter. As the attribute was specified in line 8 above the controller class name, you can revoke the filter for a specific method by using the **`AllowAnonymous`** attribute, as shown in *Figure 17.3*:

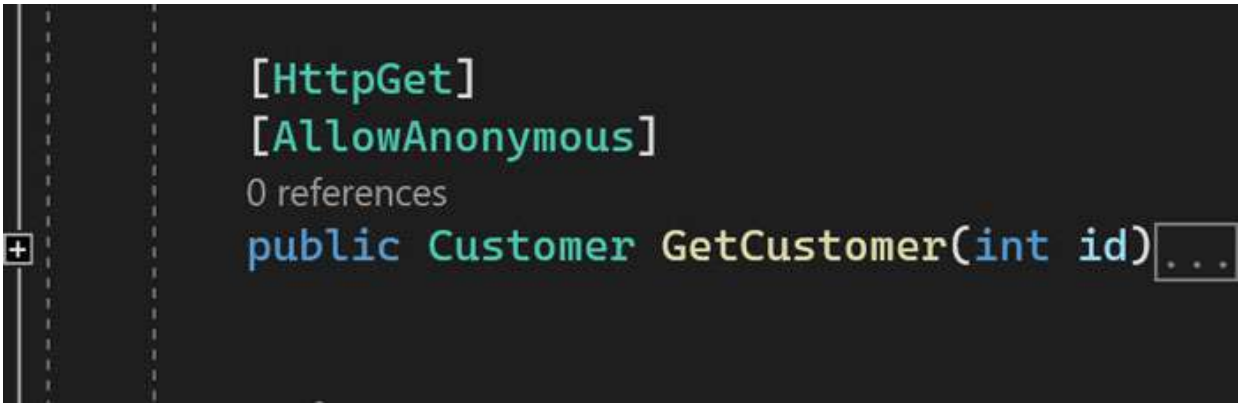


Figure 17.3: *AllowAnonymous attribute*

In terms of authorization, it is possible to specify the method and controllers restriction levels as per the user and role, as shown in *Figure 17.4*:

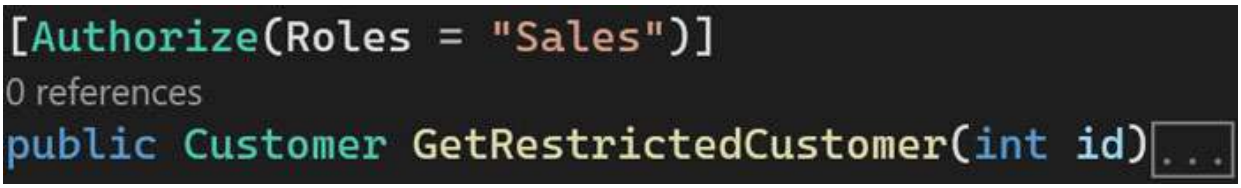


Figure 17.4: *Access restriction per role*

When there is any non-functional requirement for an application regarding authentication types, other approaches can be followed to have more sophisticated and modern authentication and authorization aspects. The next section of this chapter covers the Basic Authentication type for Web APIs.

Basic authentication

To use the Basic Authentication type for Web APIs, it is necessary to build the actual middleware that will intercept the request, read its header, and authorize or not access a specific resource via Web API. In this kind of request, the header contains the relevant information on the authentication,

such as username and password, token, or other conventions used to authorize a user or application.

The .NET platform already contains a middleware class for authentication and controller authorization based on the Authorization Filter concept. You can create your own authorization filter by inheriting the underlying base class and overriding the proper methods, as shown in *Figure 17.5*:

```
1  using System.Web.Http.Controllers;
2  using System.Web.Http.Filters;
3
4  namespace BPBWebApi.Filters
5  {
       0 references
6      public class CustomerAuthenticationFilter : AuthorizationFilterAttribute
7      {
           0 references
8          public override void OnAuthorization(HttpActionContext actionContext)
9          {
10             base.OnAuthorization(actionContext);
11         }
12     }
13 }
```

Figure 17.5: *Custom authorization filter*

In this example, the **`OnAuthorization`** method can be overridden to match a custom implementation for the authentication and authorization process, such as matching usernames and passwords with the database, applying more complex policies calling external services, and much more. *Figure 17.6* contains a representation of custom authentication logic, checking whether a username and password match a specific value:

```
public class CustomerAuthorizationFilter : AuthorizationFilterAttribute
{
    0 references
    public override void OnAuthorization(HttpActionContext actionContext)
    {
        if (actionContext.Request.Headers.Authorization != null)
        {
            var headerToken = actionContext.Request.Headers
                .Authorization.Parameter;

            var decodeHeaderToken = System.Text.Encoding.UTF8.GetString(
                Convert.FromBase64String(headerToken));

            var tokenValues = decodeHeaderToken.Split(':');

            if (tokenValues[0] == "admin" &&
                tokenValues[1] == "test")
            {
                Thread.CurrentPrincipal = new GenericPrincipal(
                        new GenericIdentity(tokenValues[0]), null);
            }
            else
            {
                actionContext.Response = actionContext.Request
                    .CreateResponse(HttpStatusCode.Unauthorized);
            }
        }
        else
        {
            actionContext.Response = actionContext.Request
                .CreateResponse(HttpStatusCode.Unauthorized);
        }
    }
```

Figure 17.6: *Custom authorization method*

In this code sample, the authorization token is read from the request header (lines 14 and 15), and the username and password are taken from the token value by splitting the content separating the two values (line 20). If the username and password match with the expected value, the **`CurrentPrincipal`** property is set with the user information to the current Thread. On the other hand, if the credentials do not match the expected values, a 401 HTTP status code is shown, which confirms to the requester that the user is not authorized.

It is essential to note in this sample that the correct username (admin) and password (test) are hard-coded, just as an example. In a real scenario, this authorization method would verify actual credentials in a database or any other authentication service. This example is kept simple to focus only on the Authorization Filter aspect for Asp.Net Core Web APIs.

To use the actual custom authorization filter, you need to specify it above a controller or method as an attribute, as shown in *Figure 17.7*:

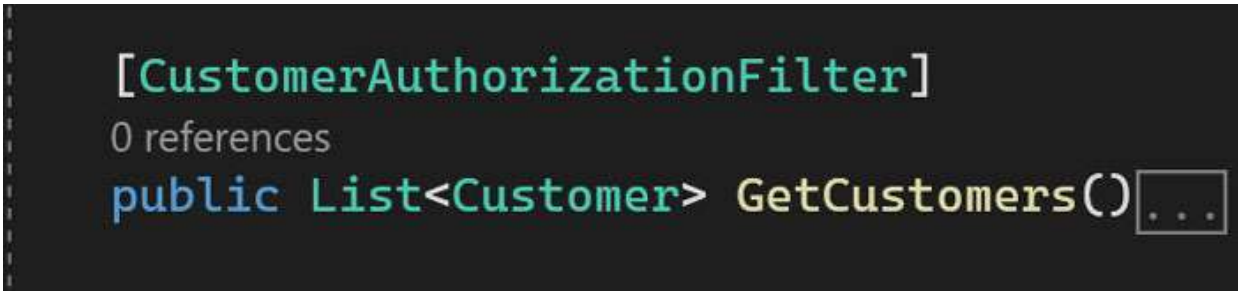


Figure 17.7: *Custom authorization filter*

Encapsulating all the logic behind authentication and authorization using authorization filters contributes to separating the responsibilities between business logic and non-functional requirements in controllers and methods.

JWT authentication

Like other technologies and platforms, Asp.Net Core projects support authentication based on JWT (JSON Web Token), which allows applications to create security mechanisms with signature and encryption associated with a request payload. In this type of authentication, the server generates a token that contains the user identity that client applications can use to send associates with requests to validate and confirm the requester's identity.

You must install Microsoft to start using JWT authentication combined with Asp.Net Core applications **`.AspNetCore.Authentication.JwtBearer`** and **`System.IdentityModel.Tokens.Jwt`** packages, as highlighted in *Figure 17.8*:

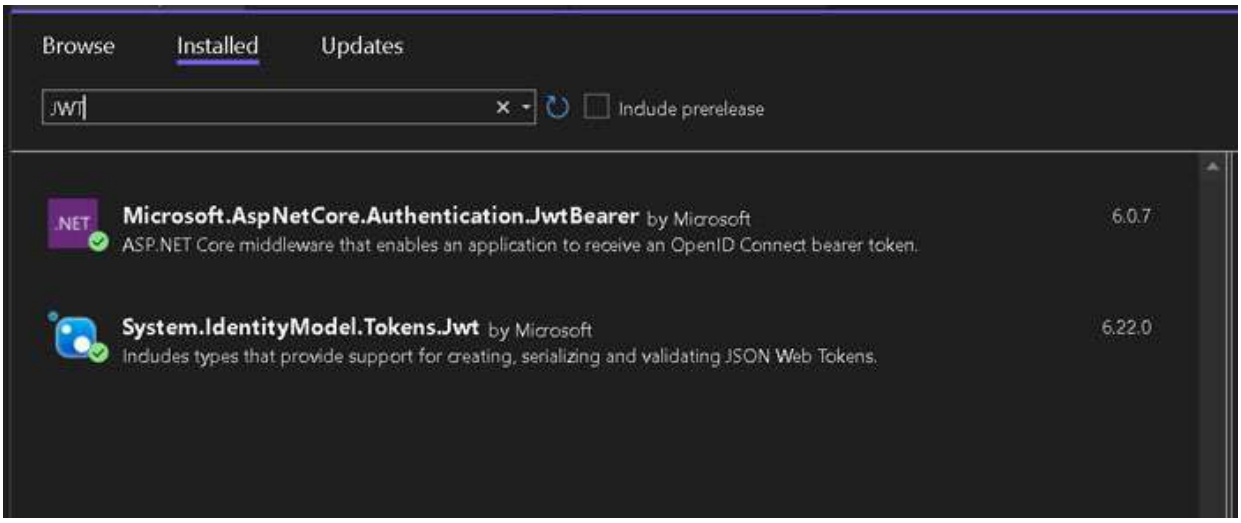


Figure 17.8: *JWT packages*

After installing the underlying NuGet packages, you need to adapt the **`appsetting.json`** file to include the keys regarding the JWT configuration, as shown in *Figure 17.9*:

Figure 17.9: *appsettings.json for JWT*

This is a representation of a JWT configuration with the following information:

- **Key**: The secret key used by the backend application to encrypt the JWT tokens. This token must be secured and complex enough.
- **Issuer:** The application or party that creates the JWT token and gives the token a private key for encryption.
- **Audience**: The server resource that should accept the token in the validation.

From Asp.Net Core 6, applications do not have the traditional **`Startup.cs`** file, instead, everything is configured for the application within the **`Program.cs`** file. Asp.Net Core follows an architecture that allows us to register services to the container pipeline. This means we can have configurations for JWT and authentication globally registered as a service for the entire application. Within the **`Program.cs`**, add the JWT configuration, as shown in *Figure 17.10*:

```
9   builder.Services.AddAuthentication(x =>
10  {
11      x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
12      x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
13  }).AddJwtBearer(o =>
14  {
15      var Key = Encoding.UTF8.GetBytes(builder.Configuration["JWT:Key"]);
16      o.SaveToken = true;
17      o.TokenValidationParameters = new TokenValidationParameters
18      {
19          ValidateIssuer = false,
20          ValidateAudience = false,
21          ValidateLifetime = true,
22          ValidateIssuerSigningKey = true,
23          ValidIssuer = builder. Configuration["JWT:Issuer"],
24          ValidAudience = builder.Configuration["JWT:Audience"],
25          IssuerSigningKey = new SymmetricSecurityKey(Key)
26      };
27  });
```

Figure 17.10: *JWT Configuration*

In this configuration, we are adding the Authentication service (line 9) and registering the JWT Bearer between lines 13 and 25. Note that the JWT settings are being taken from the **`appsettings.json`** file (key, issuer, and audience) using the **`builder.Configuration`** object. The Symmetric Security Key is being used in terms of the encryption method, but other types can be used according to convenience.

Additionally, ensure you have configured the application to use authentication and authorization within the **`Program.cs`** file, as shown in *Figure 17.11*:

Figure 17.11: *Use of authentication and authorization*

The next step is to create a class that handles the authentication process and refers to the class as a service in the **`Program.cs`** file. To achieve that correctly, we have to create three classes: user, token, and JWTAuthenticator, including the underlying interface for the authenticator.

For simplification reasons in this chapter, the user class is kept only with login and password properties, as shown in *Figure 17.12*:

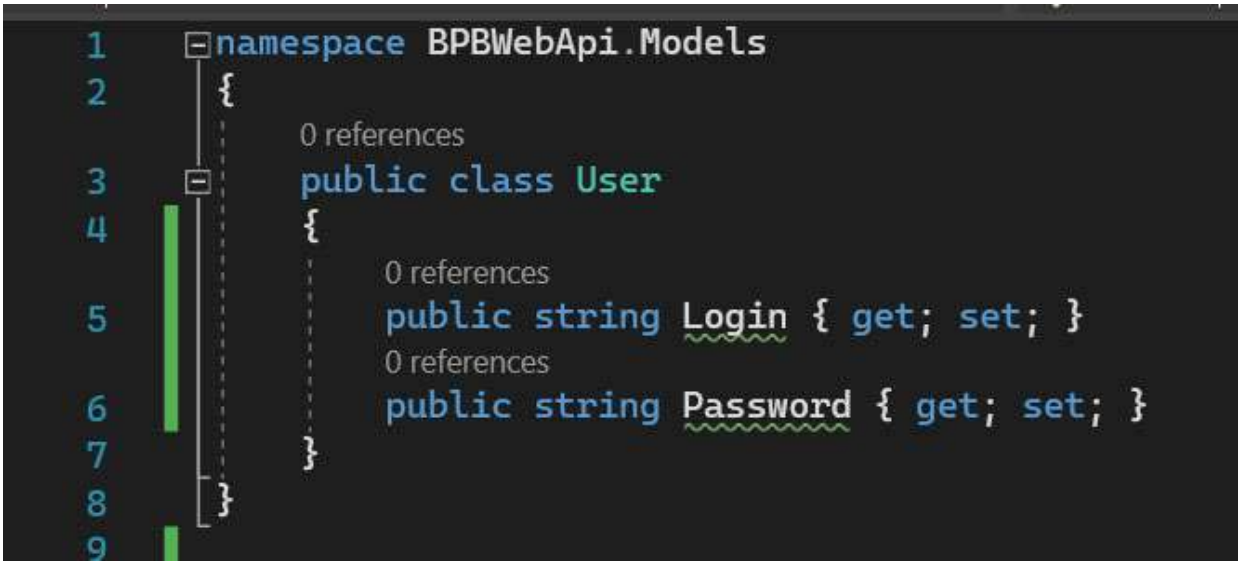


Figure 17.12: *User class*

Furthermore, the token class is represented by the code in *Figure 17.13*:

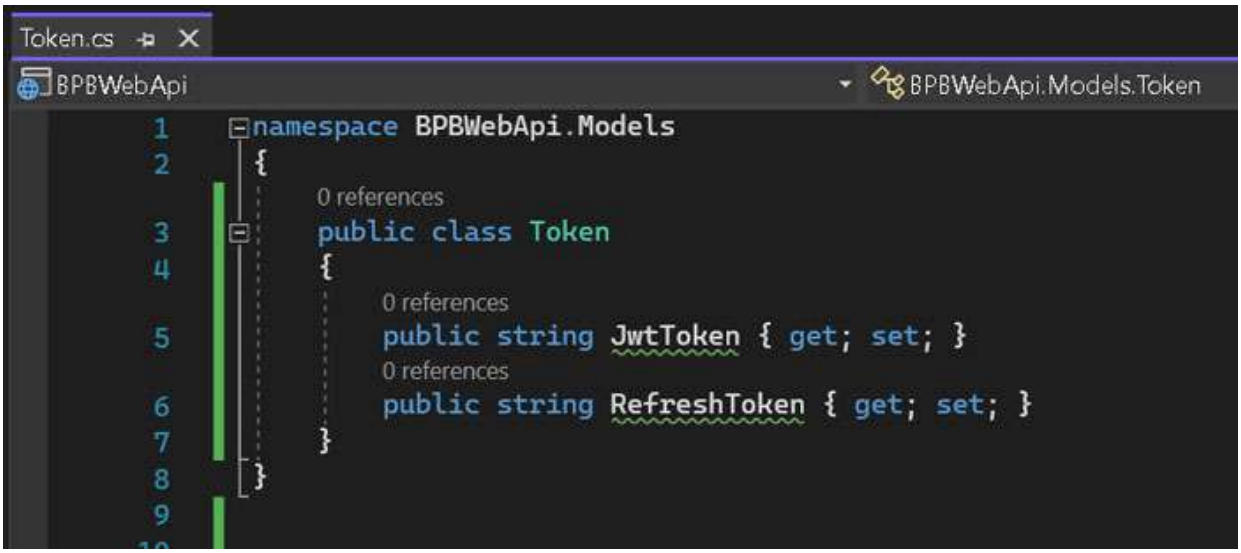


Figure 17.13: *Token class*

The first property is responsible for keeping the encrypted token returned by the authenticator, and the second property has the purpose of storing the token that can be used to get a new fresh token and renew the authentication, respecting the expiration time determined by the server application.

In the following example *Figure 17.14*, create the **`IJwtAuthenticator`**, which is going to have only one method in the specification:

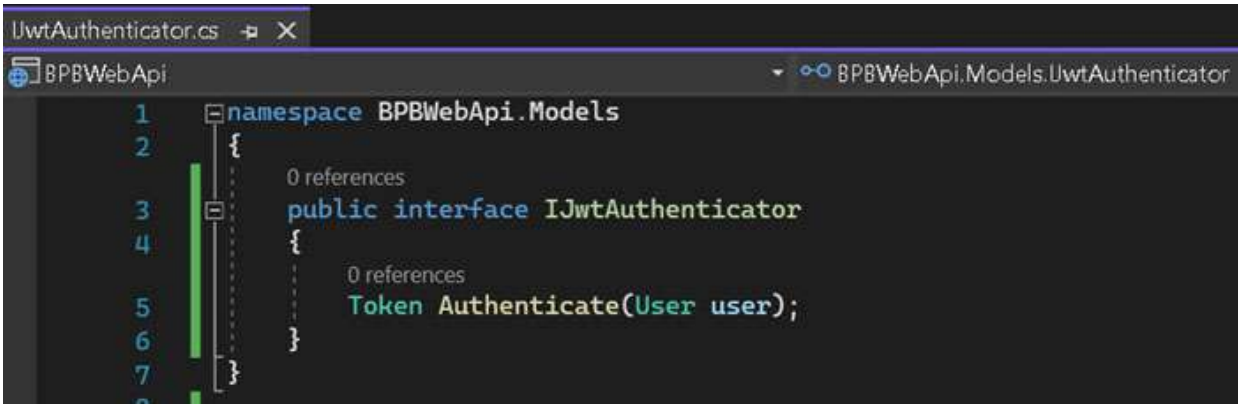


Figure 17.14: *Token class*

The implementation of the JWT Authenticator class involves including the actual logic for the authentication. For testing purposes, a fake user repository is created to hide complexities regarding a database connection. The class that needs to implement the specification of the **`IJwtAuthenticator`** interface looks like the representation in *Figure 17.15*:

Figure 17.15: *JWT Authenticator class implementation*

Between lines 10 and 15 in *Figure 15.17*, a User Repository object is created to simulate records in the database with only three users. Line 17 specifies a read-only configuration variable, which will be populated in the constructor using Dependency Injection.

After an actual implementation of the Authenticate method in this **`JwtAuthenticator`** class, it looks like the code in *Figure 17.16*:

```
1 reference
public Token Authenticate(User user)
{

    if (!UserRepository.Any(x => x.Key == user.Login && x.Value == user.Password))
    {
        throw new UnauthorizedAccessException("Invalid login and/or password");
    }


    var tokenHandler = new JwtSecurityTokenHandler();
    var tokenKey = Encoding.UTF8.GetBytes(_configuration["JWT:Key"]);
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
      {
     new Claim(ClaimTypes.Name, user.Login)
      }),
        Expires = DateTime.UtcNow.AddMinutes(10),
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(tokenKey),
        SecurityAlgorithms.HmacSha256Signature)
    };

    var token = tokenHandler.CreateToken(tokenDescriptor);
    return new Token { JwtToken = tokenHandler.WriteToken(token) };

}
```

Figure 17.16: *Authenticate method implementation*

Breaking down the Authenticate method implementation, it is possible to see the following:

- Lines 25 to 28 validate the user credentials, verifying if they match the list of users presented on the repository. The method throws an unauthorized exception if the login and password do not correspond to any value.
- Between lines 31 and 42, the **`tokenHandler`** object is created, and the token configuration is specified, assigning the JWT key from the **`appsettings.json`**, configuring the claim (user identity), and specifying an expiration for the token with the value of 10 minutes.
- Finally, line 44 creates the actual JWT token using the token description, and the method returns a Token object.

As all the necessary classes are already implemented at this point, the next step is to register the JWT Authenticator class as a service in Program.cs, as highlighted in *Figure 17.17*:

Figure 17.17: *Authenticate method implementation*

The last step in the implementation is to refer to the JWT Authentication via Dependency Injection in the controller responsible for the authentication endpoint and to expose the actual authentication method, as shown in *Figure 17.18*:

```
private readonly IJwtAuthenticator _jwtAuthenticator;
0 references
public CustomerController(IJwtAuthenticator jwtAuthenticator)
{
    _jwtAuthenticator = jwtAuthenticator;
}

0 references
public IActionResult Authenticate(User user)
{
    try
    {
        var token = _jwtAuthenticator.Authenticate(user);
        return Ok(token);
    }
    catch(UnauthorizedAccessException ex)
    {
        return Unauthorized();
    }
    catch(Exception ex)
    {
        return Problem();
    }

}
```

Figure 17.18: *JWT authenticator in the Controller*

In the controller, a JWT Authenticator object is injected automatically, and the Authenticate method receives the user information (login and password) as a parameter and calls the authenticate method from the JWT Authenticator class. If the authentication succeeds with valid credentials, an OK response is returned. Otherwise, an error is returned by the API due to unauthorized access.

Conclusion

In this chapter, you had the opportunity to learn fundamental concepts of authentication and authorization for .NET applications focused on Web Development, following good practices of security primarily used in the market, such as JWT and native features for Asp.Net Core applications.

In the next chapter, you will learn how to interact with databases using the Entity Framework Core, taking advantage of ORM capabilities in the .NET platform.

Points to remember

- It is recommended to place the connectionstring for your Azure resources into proper and secure secret settings or environment variables.
- Azure Functions support Python, Java, Node.js, .NET, and PowerShell
- It is possible to host Azure Function Apps in Linux or Windows, depending on the region selected.
- Azure SDK for Blob Storage allows us to upload, download, update, and delete files on Azure Storage Accounts, including manipulating metadata.

Multiple-choice questions

1. **Which alternative represents a false statement for Blob Storage configuration?**

    a. There is a restriction on uploading PDF files in containers.

    b. All the containers are public by default.

    c. The URL must be unique.

    d. It is possible to configure access control per container.

2. **Which .NET versions are supported for Azure Functions?**

    a. .NET Framework 2.0, .NET Core 1.0, .NET Core 3.1, and .NET 5

    b. All .NET Core versions

    c. .NET 5 and .NET 6

    d. .NET Core 3.1 and .NET 6

3. **Which service is not supported by Azure Storage Accounts?**

    a. Tables

    b. File Share

c. Static website
d. SignalR

Answers

1. **b**
2. **d**
3. **d**

Questions

1. Explain what is a serverless application.
2. Create an Azure Function that accepts only POST HTTP Verb and returns a **querystring** parameter.
3. Create a Console App that uploads a file into an Azure Blob Storage container and downloads the same file into a local folder.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 18

Introduction to Entity Framework Core

Introduction

Interaction with databases is one of the most significant tasks in software development as most of the applications that have some level of user interfaces are, in general, interfaces to allow users to manipulate information stored in a database by performing CRUD operations: create, read, update, and delete data. Integration with databases to perform these operations is not a trivial task. Entity Framework Core helps .NET developers to simplify this process using a mature **Object Relational Mapping** (**ORM)** framework.

Learning Entity Framework Core will allow you to build robust applications interacting with databases using C# language, speeding up the development process in terms of productivity while taking advantage of all the existing benefits of the object-oriented programming paradigm.

This chapter will cover the essential aspects of **Entity Framework Core** (**EF Core**), including basic concepts of ORMs in general, **Language Integrated Query** (**LINQ)**, and how to use EF Core with SQL Server database.

Structure

In this chapter, we will discuss authentication and authorization for Asp.Net Core applications on the following topics:

- Basic concepts of ORM
- Entity Framework Core
- Introduction to LINQ

Objectives

After studying this unit, you should be able to understand the basic concepts of ORM, build simple applications for CRUD operations, apply LINQ to perform database queries using C#, and understand the good practices of EF Core.

Object Relational Mapping (ORM)

Almost all the enterprise systems that all developers work in daily involve the need to implement primary operations in databases, which implies using techniques to interact with databases using one of the multiple programming languages. This chapter aims to show you how Entity Framework Core works; however, it is crucial to walk you through some basic concepts of Object Relational Mapping, which is the foundation of Entity Framework Core and similar frameworks.

Object Relational Mapping and **Object-Oriented Programming** (**OOP**) are directly related concepts. The use of OOP is significant because the code becomes closer to real-world concepts, becoming easier to understand and interpret when any maintenance needs to be applied by developers. For example, if we are developing a basic system to register Students, the underlying simplified C# class for this student would look like as shown in *Figure 18.1*:

```
Student.cs
EntityFrameworkCoreConsole                    EntityFrameworkCoreConsole.Student
1   namespace EntityFrameworkCoreConsole
2   {
        0 references
3       public class Student
4       {
            0 references
5           public int Id { get; set; }
            0 references
6           public string Name { get; set; }
            0 references
7           public string Email { get; set; }
            0 references
8           public DateTime BirthDay { get; set; }
9       }
10  }
```

Figure 18.1: *Student class*

Note that this class is quite similar to how we would describe a student in the "real world." In a conventional coding routine, the full implementation of this class may involve different methods for making the most basic operations that an actual system would perform: list students, register students, update students, and much more.

Almost all these operations require the developer to create an interface with a database using C# or any other programming language based on the OOP paradigm. For this, the developer would need, programmatically, to open a connection with the database to perform basic operations and SQL commands: SELECT, UPDATE, DELETE, and so on.

This is a pretty common scenario, and these operations may not be so trivial as developers would need to write all the **Structured Query Language** (**SQL**) instructions manually, write the C# code related to the classes and databases operations, and parse manually the information brought from the database to C# objects, populating it object property individually, mapping it with database tables and columns, even database objects, and C# objects don't necessarily have a genuine correspondence. It is possible to say that, in the context of a system that needs to manage Student data, the columns on a database table would have the representation shown in *Figure 18.2*:

Student		
Id	int	Primary Key
Name	varchar(100)	
Email	varchar(100)	
Birthday	datetime	

Figure 18.2: *Student table*

The columns for the database table are similar to the Student class properties. Each database has its types: SQL Server, Postgres, Oracle, and so on. Therefore, the representation of the student table may vary depending on

which database is used. However, although the properties in the C# class and the table columns are almost identical, a developer would need to write a relevant amount of code to connect to the database, get the table records, and populate C# objects programmatically to integrate properly an enterprise application built using the .NET platform and the database provider.

Additionally, extra database commands would need to be written and referenced in the C# program to apply typical commands to handle data via application: insert, update, and delete operations. Even for systems considered small or with low complexity, the database integration might be repetitive and exhausting work that may need to be applied hundreds or thousands of times depending on the number of entities an application needs to manage.

The challenge in this context is the incompatibility between C# classes and database structure. Classes follow the object-oriented programming paradigm and databases follow another approach, such as a relational or non-relational model. Therefore, applications and databases are entirely different things.

The market has a consolidated model to combine object-oriented programming with relational databases to solve this problem: the **Object Relational Mapping** (**ORM**) paradigm. As its name states, ORM is a technique that facilitates the mapping between classes and relational databases, allowing direct correspondence between them. Many ORMs are available in the market, like Hibernate, Entity Framework, and Dapper for the .NET platform.

There are pros and cons to using ORM tools in general, and the discussion around this topic remains open after many years, mainly in terms of performance. When an ORM is used, the developer transfers the definition of database queries to the ORM. The ORM might not perform complex operations well when aggregation and other commands involve many nested database calls. However, if the use of ORMs is combined with a good understanding of relational database queries, there is a good chance of satisfactory performance for most scenarios.

Entity Framework Core

Entity Framework Core is the most popular ORM used by .NET developers. The tool is open-source and maintained by Microsoft and contributors from

the community. Microsoft released the first version of Entity Framework in August of 2008, part of the .NET 3.5, while Visual Studio 2008 was launched. The first versions were called only “Entity Framework” and not Entity Framework Core, as a re-branding happened after Microsoft released the .NET Core 1.0.

As any ORM has the purpose of correlating classes and tables, there are two ways to achieve that using Entity Framework Core, as follows:

- **Code first**: In this model, the C# classes are created first, and the composition of the classes is used to generate the database structure.
- **Database first**: It is possible to reverse the database structure and create the C# classes following the definition present in the database structure.

In the context of this book, the code-first approach is used along with all the code samples in the following sections. To demonstrate the creation of a database via code using Entity Framework Core, imagine a hypothetical scenario where you have to build an application to store customers using C# language. For this, it is fundamental to define two main things:

- Customer class in a C# project
- Database with the underlying table

In the Code first approach, the first thing that needs to be done is the Customer class. In a simplified way, this class would look like *Figure 18.3*:

Figure 18.3: *Customer class*

For this example, a simple Console Application in C# can be used to test the Entity Framework configuration and database interactions. After creating the Console App in Visual Studio, install the following NuGet packages presented in *Figure 18.4*:

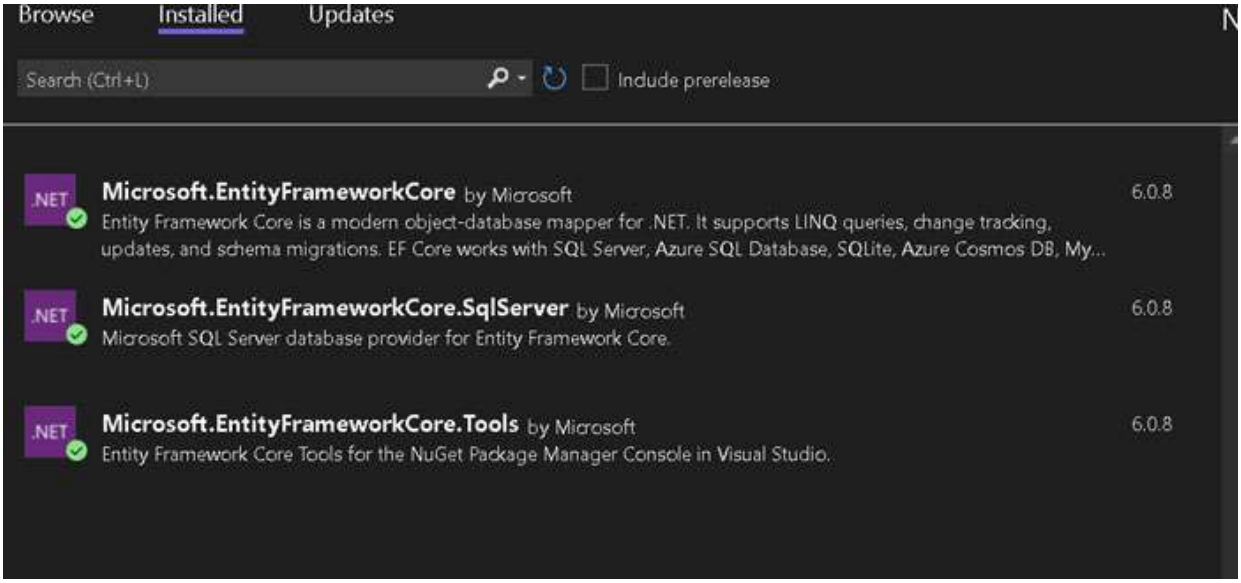


Figure 18.4: *NuGet packages for Entity Framework Core*

After installing the packages and creating the Customer class in the Console project, it is necessary to configure Entity Framework to target a specific

database. Entity Framework Core uses the concept of **Migrations,** which are descriptive database instructions written in C# to allow developers to check the structural changes that Entity Framework will apply to the database itself, following the C# models configuration that is specified by the application as part of the Entity Framework context.

The Entity Framework Core package contains a **`DbContext`** class that can be used to specify details on the application classes that the ORM should consider to map to database objects. Therefore, you have to create the underlying **`DbContext`** for the sample project, referring to the Customer class, as shown in *Figure 18.5*:

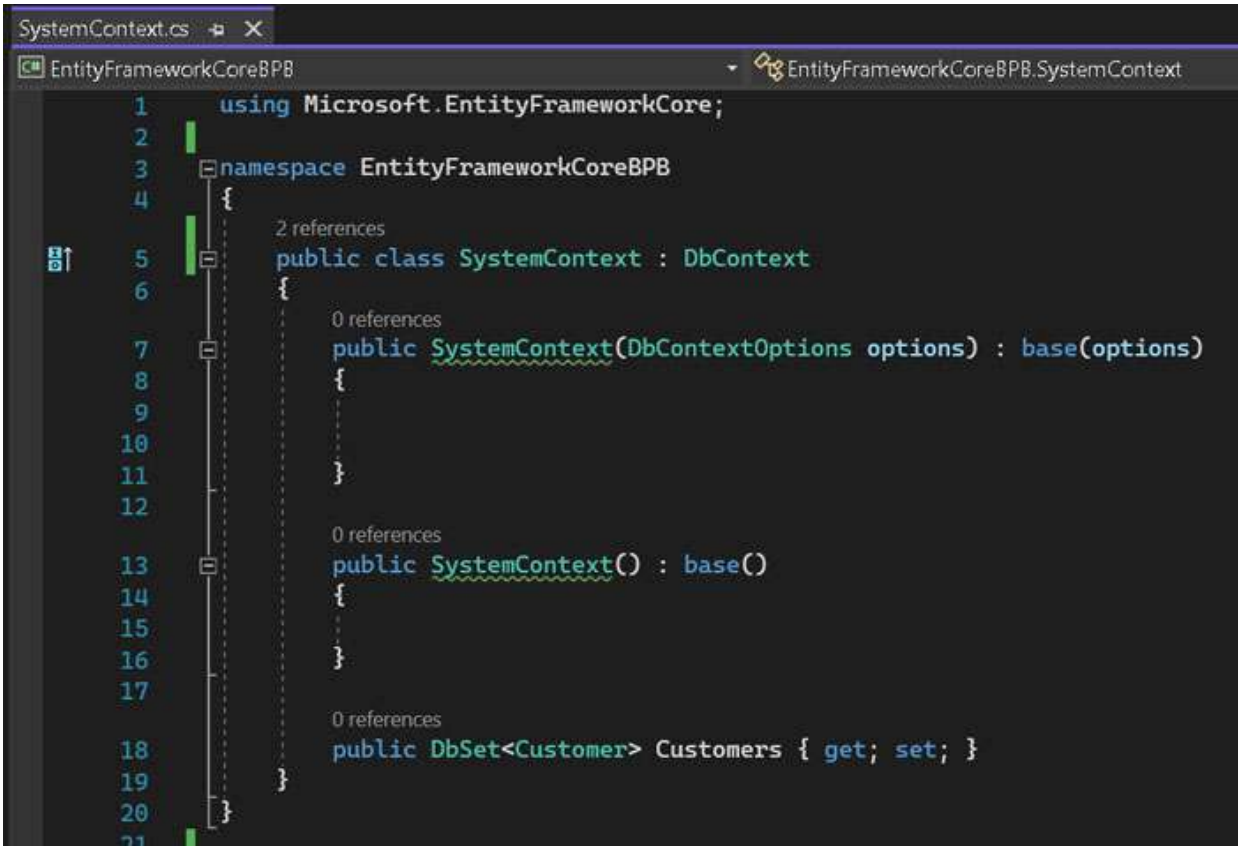


Figure 18.5: *SystemContext class*

As seen in *Figure 18.5*, a **`SystemContext`** class inherits from the **`DbContext`** class, having a single property to represent a **`DbSet`** for the Customer class. This configuration makes explicit to Entity Framework the intention of mapping the Customer class with a database table. The **`DbSet`** class from the Entity Framework Core package allows us to manipulate the data using LINQ queries. It contains methods to add, update, and delete customers from

the database as well, converting LINQ queries automatically to database commands.

The application must be configured to point to a specific database containing the sample application's customer table. In the context of this chapter, the SQL Server Express version is being used, which is a free version available for studying purposes. The connection string required for the database authentication needs to be placed in the system context class within the **`OnConfigure`** that is overridden from the base **`DbContext`** class, as shown in *Figure 18.6*:

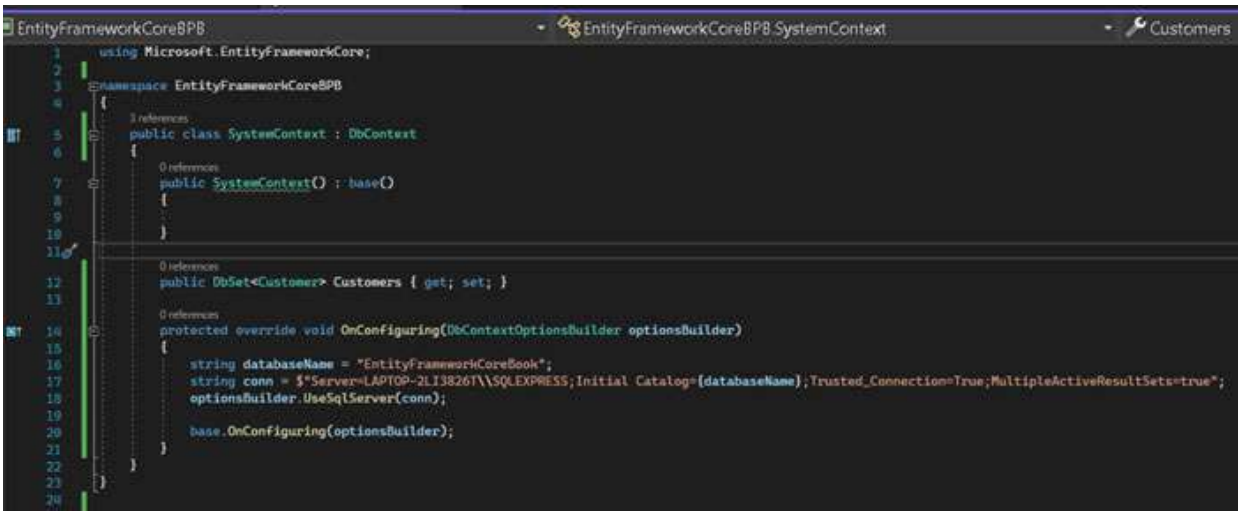


Figure 18.6: *Database connection configuration*

As shown in *Figure 18.6*, line 16 specifies the database name, and line 17 contains the path to a local SQL Server Express database. To create the example yourself, you have to replace the database path to your instance of SQL Server, which could be a local installation or a database server presented remotely.

Once you complete the database configuration, including creating the Customer and System Context classes, you can run the command “add-migration” followed by the name desired for the migration. This command can be executed in the Package Manager Console panel and compares the database structure with the models present in the **`SystemContext`** class. If the database and underlying do not exist yet, the “add-migration” command generates a file that contains instructions to create the related objects into the database. Once you run the “add-migration” command, you should see the result as shown in *Figure 18.7*:

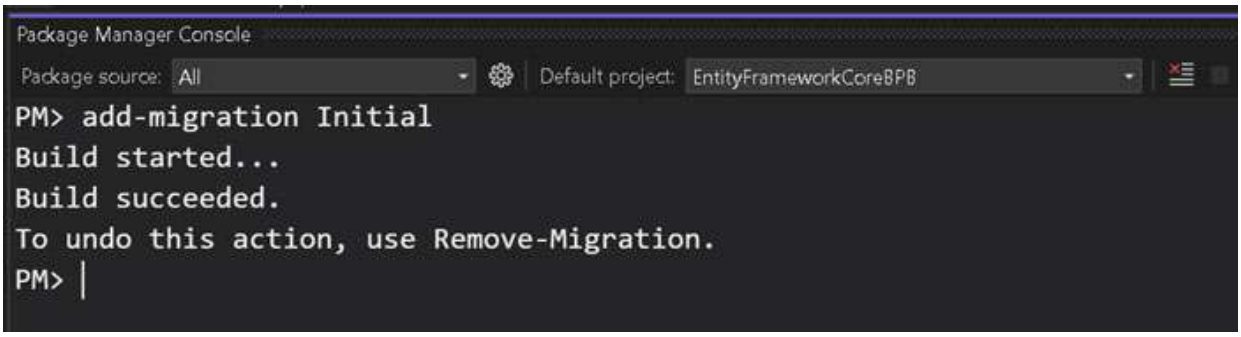


Figure 18.7: *Add-migration command*

After running the command, a Migrations folder is created, and it is visible in the Solution Explorer, as shown in *Figure 18.8*:

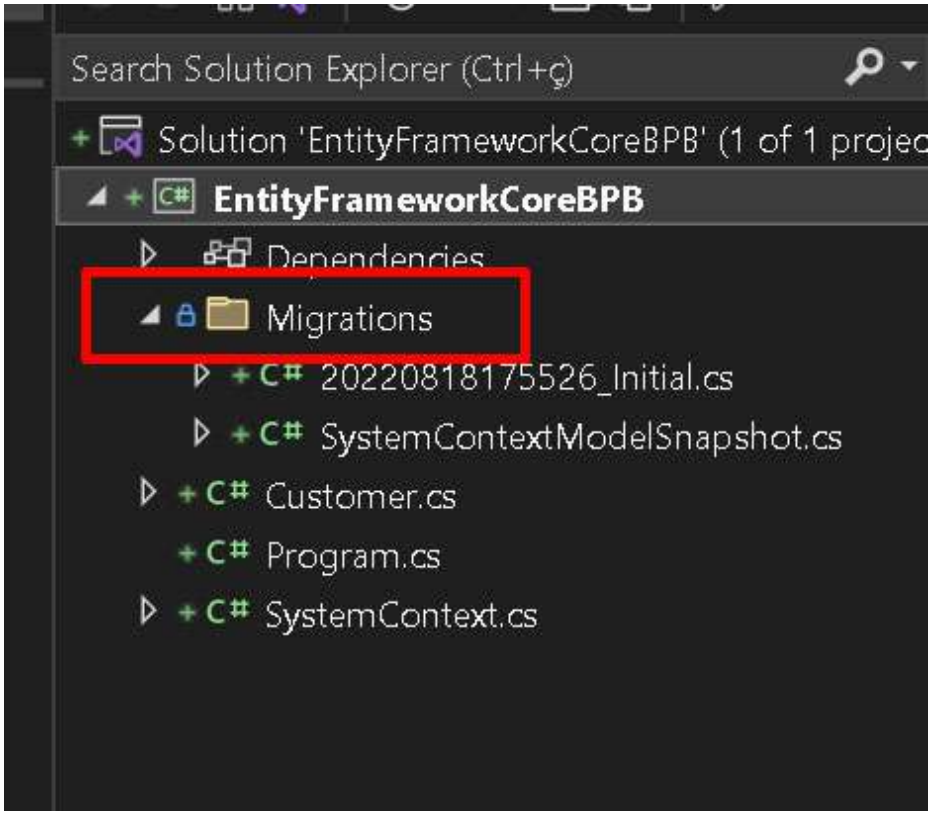


Figure 18.8: *Migrations folder*

At this stage, the actual database is not created, only the migration routine containing the specification to execute the underlying database script by Entity Framework. Under the Migrations folder, you will see the migration file containing the name given to the migration when the "**`Add-migration`**" command is executed (Initial in this context), followed by the timestamp

regarding the exact moment the migration was generated. *Figure 18.9* shows what the migration file looks like:

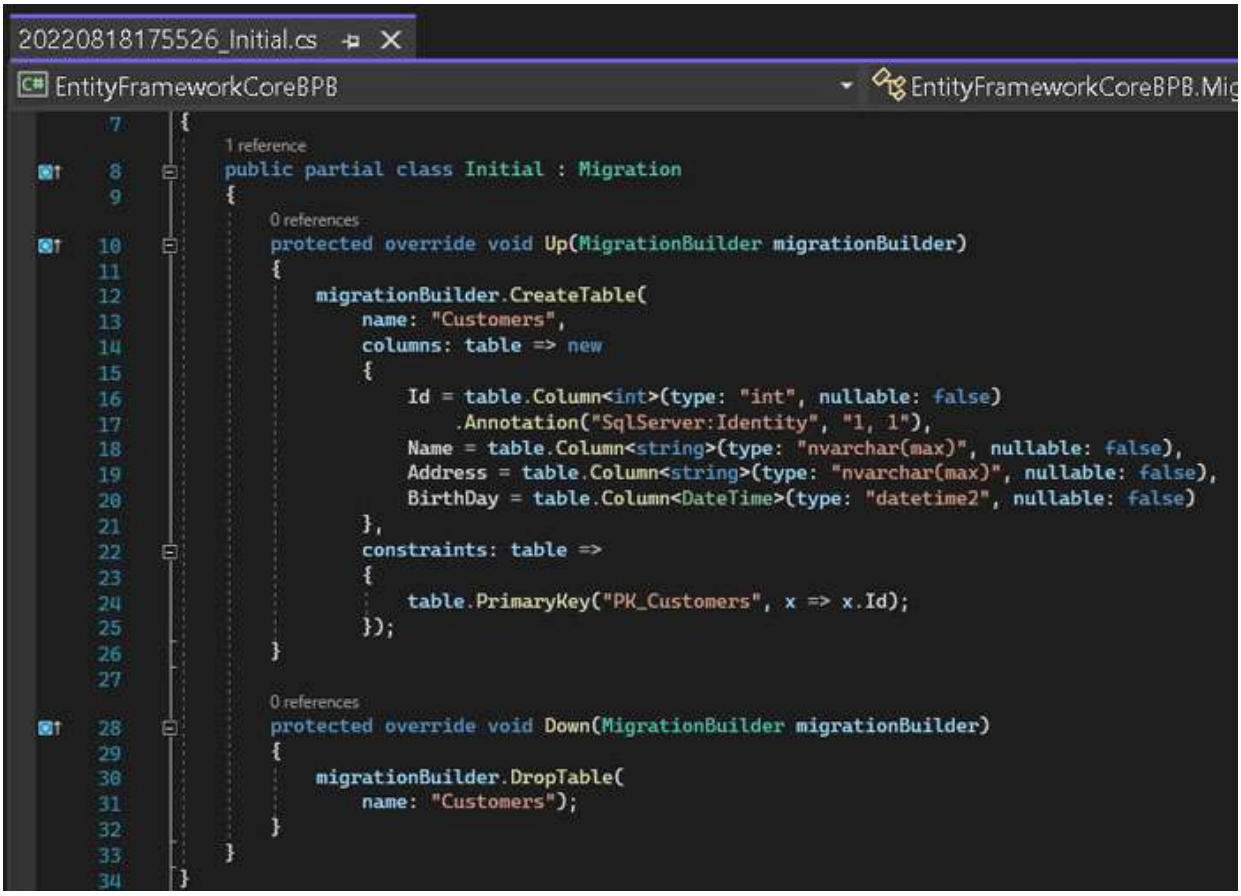


Figure 18.9: *Initial migration*

As seen in *Figure 18.9*, the **`CreateTable`** is called receiving as a parameter the list of columns that need to be created in the database: ID, Name, Address, and Birthday. The Entity Framework is smart enough to correlate the C# types with the database field types. It is possible to customize the types for each property. Still, using the default types provided by Entity Framework is the most common thing done by many companies over the years and usually meets the most basic requirements for regular enterprise applications. The migration file represents an opportunity for developers to check what Entity Framework will execute in the database to see if the proposed database changes represent the expected result.

Once an Entity Framework migration is created, you can run the “Update-database” command on the Package Manager Console, which will effectively apply the changes to the database, creating the necessary objects

or modifying existing ones if a migration is referring to an existing database that requires changes. After the “Update-database” command is executed, a successful message should be presented in the Console, as shown in *Figure 18.10*:

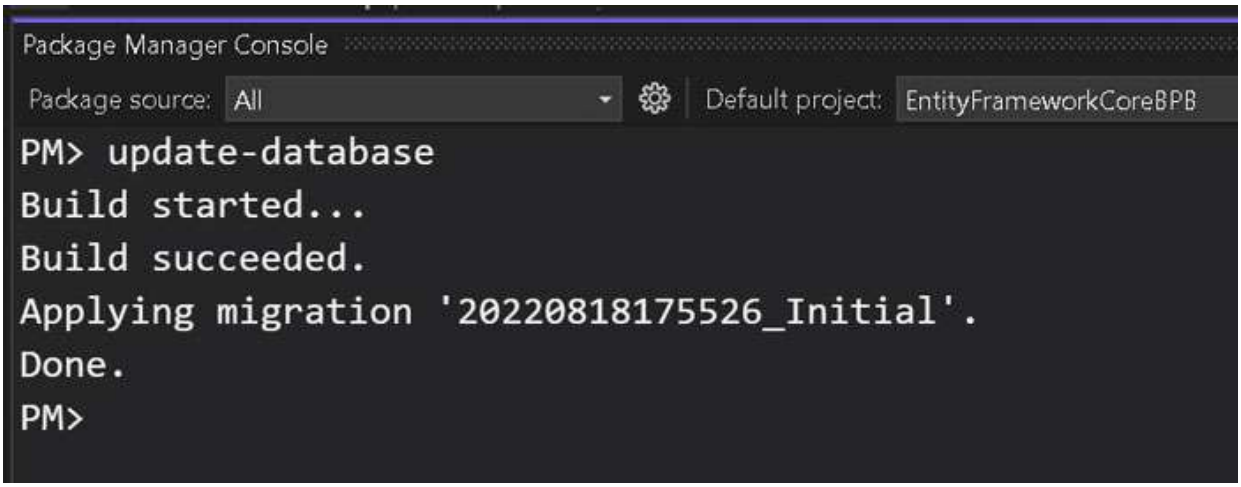


Figure 18.10: *Update-database command*

After running the “update-database” command, the underlying objects are created in the database, as can be seen in the SQL Server Management Studio or any tool you have available to manipulate databases directly, as shown in *Figure 18.11*:

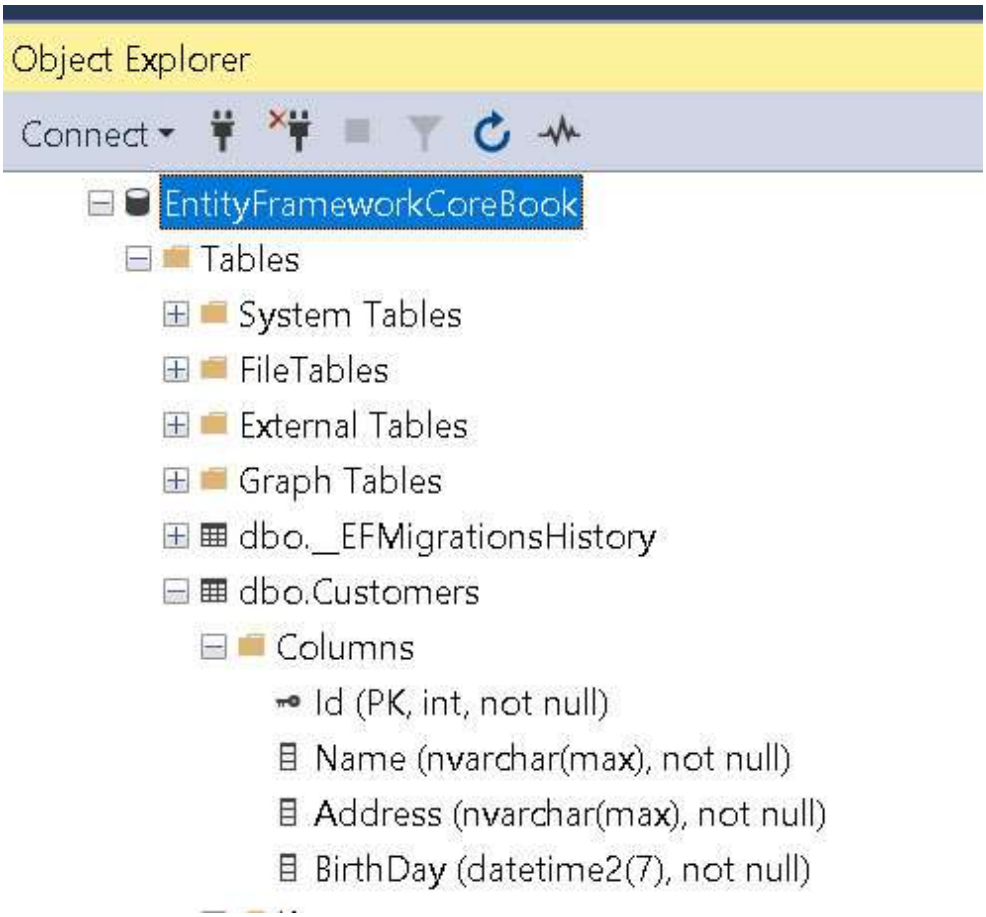


Figure 18.11: *Database creation*

The database "**`EntityFrameworkCoreBook`**" was created, respecting the same configuration presented in the database connection configuration on the **`SystemContext`** class. Additionally, the Customer table was created based on the class properties, with the ID property as the Primary Key. Suppose no explicit primary is specified in the C# class. In that case, the default configuration for Entity Framework assumes that if the class has a property called "ID," that can be considered as the primary key for the corresponding table.

Considering the "Database first" approach is being used, any changes in the database structure must be done via code, including creating new tables, columns, and other objects. To test how Entity Framework handles database changes, you can create an extra property called "DocumentNumber" in the Customer class, as shown in *Figure 18.12*:

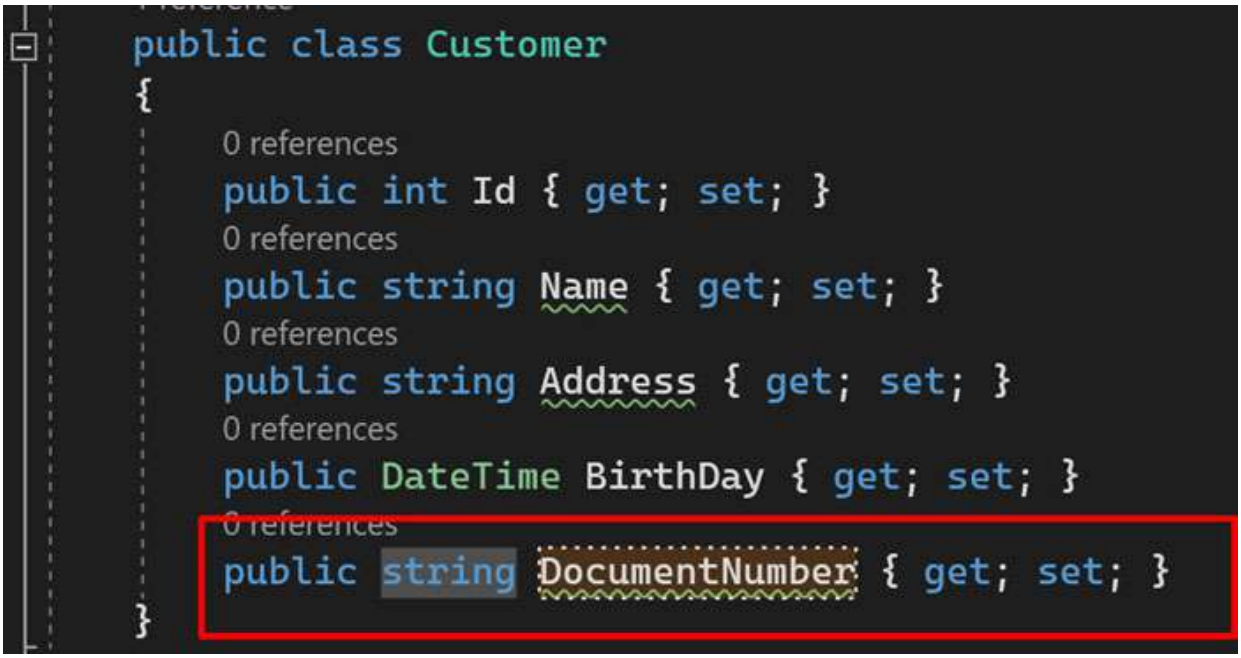


Figure 18.12: *Document number property*

In this case, as the database table and the Customer class are no longer compatible, a new migration must be created to contemplate this change. If you run the "Add-Migration" command again, you will see that a new migration is created, referring to the column change for the **`DocumentNumber`** property. If the Entity Framework command succeeds, the result on the Package Manager Console will look like *Figure 18.13*:

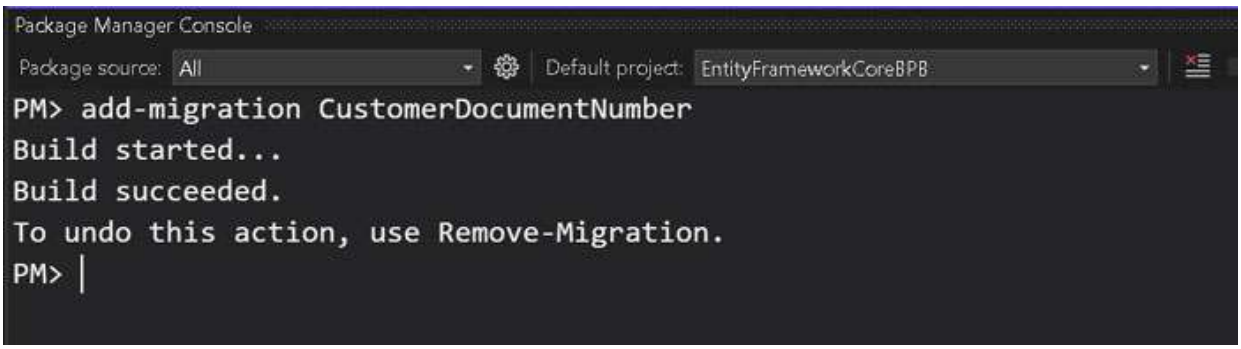


Figure 18.13: *Add-migration for Document Number*

Every time the migration is created, a new file is generated under the Migrations folder, containing details on the changes, as shown in *Figure 18.14*:

```
namespace EntityFrameworkCoreBPB.Migrations
{
    1 reference
    public partial class CustomerDocumentNumber : Migration
    {
        0 references
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "DocumentNumber",
                table: "Customers",
                type: "nvarchar(max)",
                nullable: false,
                defaultValue: "");
        }

        0 references
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "DocumentNumber",
                table: "Customers");
        }
    }
}
```

Figure 18.14: *Migration for Document Number*

All the migrations have two principal methods: Up and Down. The first one is the actual operation to apply the necessary changes and the second method is to roll back any changes if the database needs to go back to the previous state. Therefore, the Down method has the opposite operation compared to the Up method. Remember that after the creation of migrations, you need to run the “update-database” command to apply the actual changes in the database.

Entity Framework Core allows us to create more complex database structures involving multiple tables with various types of relations between them:

- One-to-one
- One-to-many
- Many-to-many

These types of relationships between tables are essential for an efficient database model. These relations can be specified at the code level by

creating relations between the C# classes. To have a more complex example, imagine the following database model demonstrated in *Figure 18.15*:

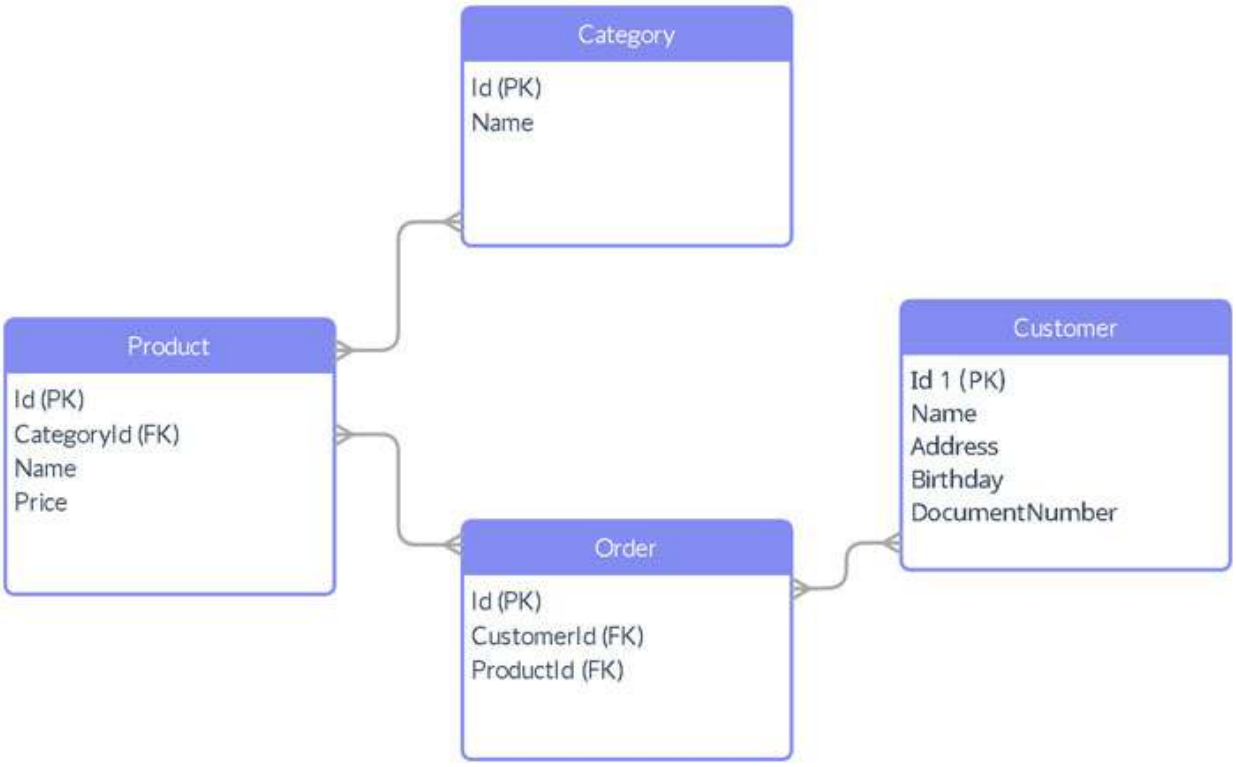


Figure 18.15: *Database model*

In the database model proposed, a Product table is associated with the Category table, as a product must have a category. On the other hand, the Order table represents a many-to-many relationship involving the Product and Customer tables. This model is particularly useful to demonstrate how we can specify these same relationships while creating C# models used by Entity Framework Core.

To translate this model to a C# class representation, create a Category class in your project, as shown in *Figure 18.16*:

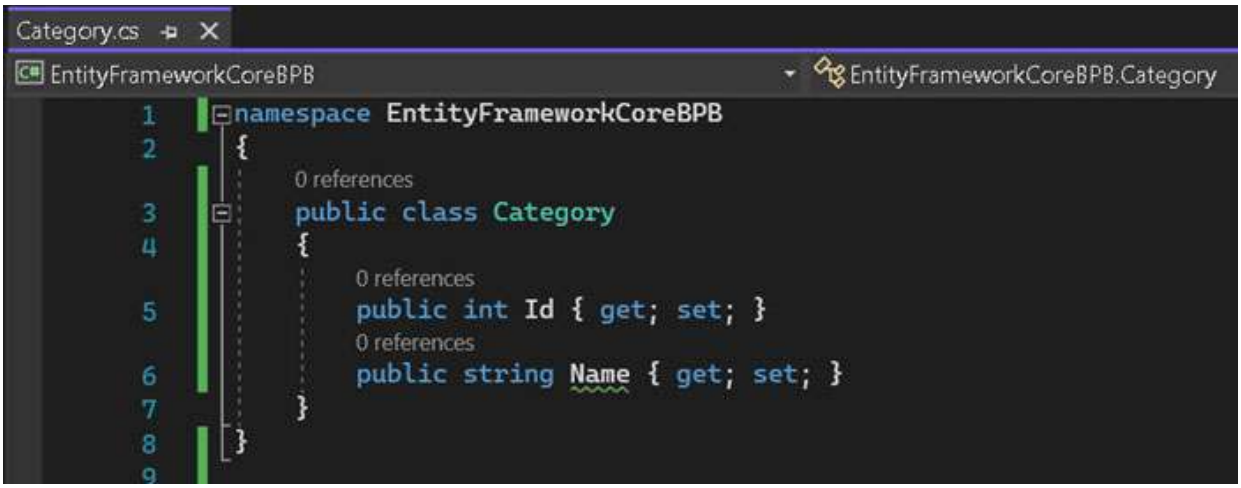


Figure 18.16: *Category class*

Note that the properties for the Category class are similar to the columns for the Category table in the proposed model. The next step is to create the Product class, following the exact representation of the Product table, as shown in *Figure 18.17*:

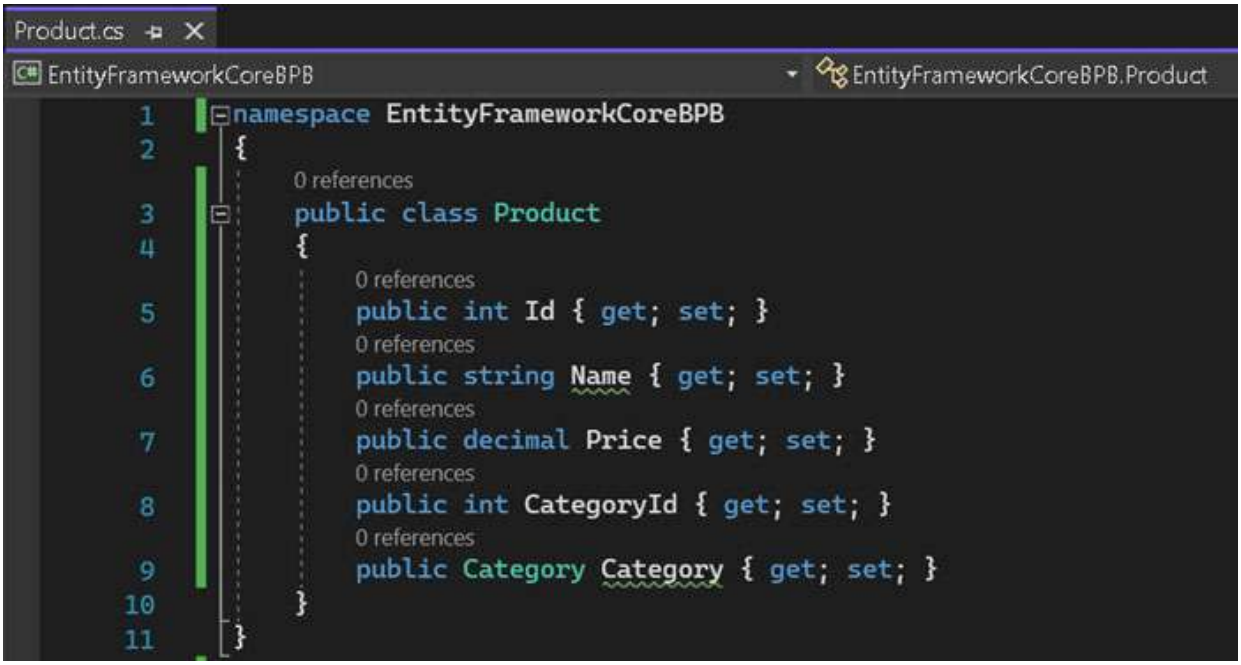


Figure 18.17: *Product class*

For the Product class specification, there is a property called "**`CategoryId`**" and another from the Category type. The Entity Framework will use these two references to correlate the two entities (Product and Category), create a

Foreign Key in the database for the Product table, and refer to the Category table.

For the Order model, the process is similar: a reference to the Customer and Product models needs to be present, as shown in *Figure 18.18*:

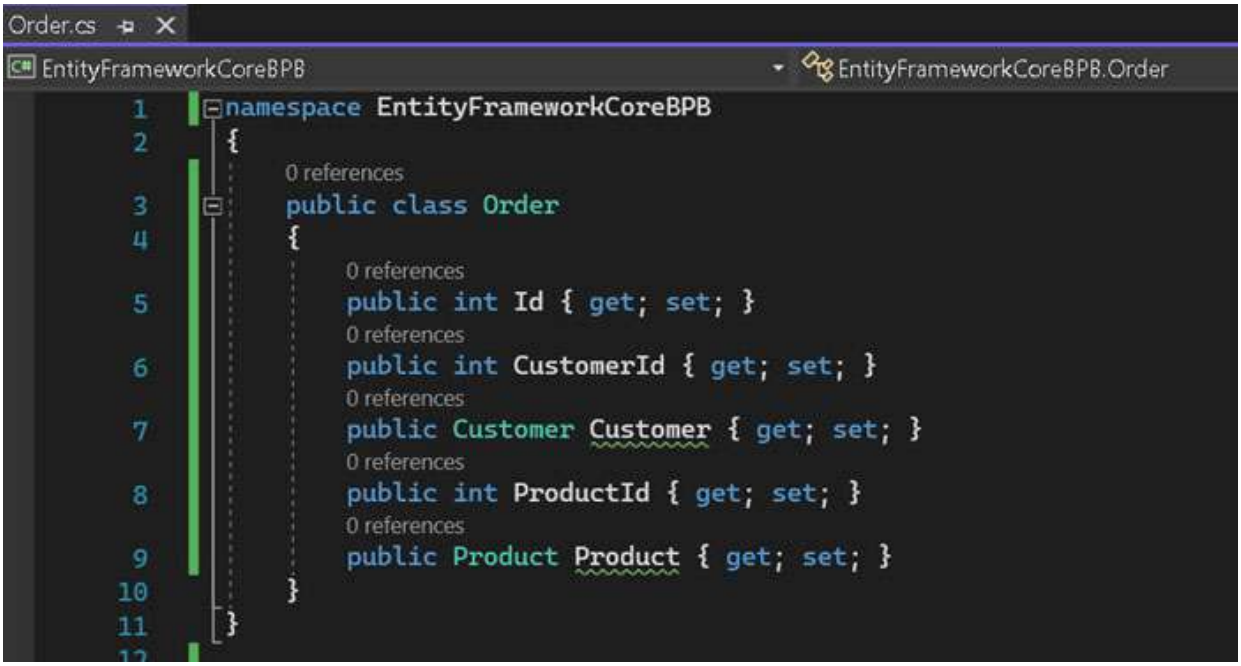


Figure 18.18: *Order class*

After creating all the classes, it is necessary to create the underlying properties in the **`SystemContext`** class for the new classes recently created: Product, Category, and Order, as highlighted in *Figure 18.19*:

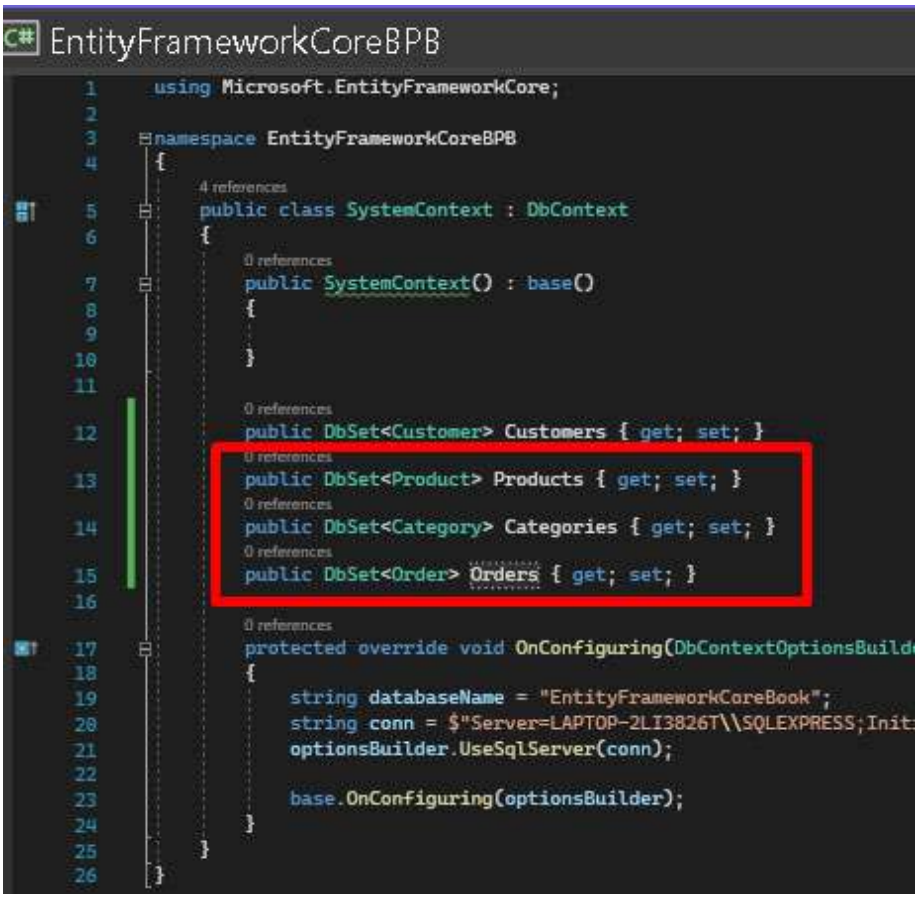


Figure 18.19: *SystemContext changes*

Considering all the relevant classes are already created at this stage, and the necessary changes to the **`SystemContext`** class were already applied, the next step is to run the "Add-migration" command again and check the result in the database structure. After running the command, you should see a migration file with a specification of the three new tables, including the configuration of the foreign keys involving the relevant tables, as shown in *Figure 18.20*:

```
migrationBuilder.CreateIndex(
    name: "IX_Orders_CustomerId",
    table: "Orders",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_Orders_ProductId",
    table: "Orders",
    column: "ProductId");

migrationBuilder.CreateIndex(
    name: "IX_Products_CategoryId",
    table: "Products",
    column: "CategoryId");
```

Figure 18.20: *SystemContext changes*

If the migration was generated with the expected changes, you could apply the changes to the database by running the "update-database" command one more time. At this point, all the necessary database objects for this example are already created, and it is possible to start manipulating data using Entity Framework.

The **`DbContext`** class contains methods to add, update, and delete records from the database. To add a new entity into the database via Entity Framework, you can populate an object and use the related **`DbSet`** to access the **`DbContext`** methods for the class, including the Add method, as shown in *Figure 18.21*:

```
1   using EntityFrameworkCoreBPB;
2
3   Customer customer = new Customer();
4   customer.Name = "Customer test";
5   customer.BirthDay = new DateTime(1987, 01, 01);
6   customer.Address = "Test street at Test city";
7   customer.DocumentNumber = "1234567";
8
9   SystemContext systemContext = new SystemContext();
10
11
12  systemContext.Customers.Add(customer);
13
14
15  systemContext.SaveChanges();
16
```

Figure 18.21: *SystemContext changes*

As highlighted in *Figure 18.21*, a Customer object is populated, and a new instance of **`SystemContext`** class is created to allow operations regarding customer entities in general: add, update, or delete. After placing this code into the **`Program.cs`** file, execute the Console application. You can verify that a new record is generated in the database in the Customer table, as shown in *Figure 18.22*:

```
SELECT TOP (1000) [Id]
      ,[Name]
      ,[Address]
      ,[BirthDay]
      ,[DocumentNumber]
  FROM [EntityFrameworkCoreBook].[dbo].[Customers]
```

150 %

Results | Messages

	Id	Name	Address	BirthDay	DocumentNumber
1	1	Customer test	Test street at Test city	1987-01-01 00:00:00.0000000	1234567

Figure 18.22: *SystemContext changes*

Entity Framework encapsulates many complexities involving database operations: connection, transaction, commits, and other relevant operations. As demonstrated for the Add method, a similar process can be done to remove records from the database and update entities. The way is the same by calling the underlying method.

LINQ

LINQ stands for Language-Integrated Query, one of the most valuable features in C# language to manipulate Lists and other data structure types. Entity Framework Core uses LINQ as a wrapper to build the database queries. Therefore, C# developers can specify complex database queries only using C# language and object-oriented programming paradigms.

Continuing with the example given in this chapter on the Product, Customer, Order, and Category classes, it is possible to get information from the database only specifying LINQ queries associated with the **`DbSet`** objects for each entity. For instance, the LINQ query demonstrated in *Figure 18.23* shows how to retrieve from the database the customer that has ID number one:

```
1  using EntityFrameworkCoreBPB;
2
3  SystemContext systemContext = new SystemContext();
4  var customer = systemContext.Customers
5      .Where(x => x.Id == 1)
6      .FirstOrDefault();
7
```

Figure 18.23: *LINQ query*

In this example, the Where method is being used, and a lambda expression is passed as an argument to determine the query's condition. The **`FirstOrDefault`** method means that a single record should be returned from the query, and a null Customer object should be returned if the query does not match any results from the database. Entity Framework Core translates the LINQ queries to database queries, facilitating the integration between the application and the database.

Furthermore, extra database operations can be specified using LINQ, such as ordering, aggregation, and much more. The following example in *Figure*

18.24 demonstrates how to retrieve a list of customers from the database, ordered by customer name:

```
2
3   SystemContext systemContext = new SystemContext();
4
5   List<Customer> customers = systemContext.Customers
6       .OrderBy(x => x.Name)
7       .ToList();
8
```

Figure 18.24: *LINQ query for Lists*

Regular database queries in real systems involve a relevant number of database queries using INNER JOIN, LEFT JOIN, and RIGHT JOIN clauses. Entity Framework Core allows us to create the same commands using LINQ, as represented in *Figure 18.25*:

```
1   using EntityFrameworkCoreBPB;
2   using Microsoft.EntityFrameworkCore;
3
4   SystemContext systemContext = new SystemContext();
5
6   List<Product> products = systemContext.Products
7       .Include(x=> x.Category)
8       .ToList();
9
```

Figure 18.25: *Inner Join using LINQ*

The Include method can be used to make inner join sentences for LINQ queries. In the last example, the query returns a list of products with the category property populated. A database query would represent a SELECT statement in the product table with an inner join statement to the category table.

LINQ is a productive feature in C# to manipulate objects, lists, and other types of data structure, allowing developers to be focused more on programming language than SQL statements. This feature applies to Entity Framework Core and C# language in general. If LINQ statements are used in the context of a **`DbSet`** object, commands to the database will be applied. If

no `DbSet` is involved, it will represent only the application's data manipulation in memory. There are a considerable number of operations that can be done using LINQ. The purpose of this book is to give you awareness of this feature. For more detailed information, please consult the official documentation.

Conclusion

In this chapter, you had the opportunity to learn the most basic concepts of Entity Framework Core, including a practical exercise on integrating a simple Console application to a SQL Server database by applying the Code First approach and LINQ statements.

Entity Framework Core is one of the most used features in .NET applications, as most enterprise systems communicate with one or multiple databases. The performance of queries generated by Entity Framework has improved on each .NET release, having its prime in the .NET 6 release. Therefore, learning more deeply about this vital framework is key to a more productive development process using this powerful ORM.

In the next chapter, you will learn relevant general good practices for .NET applications.

Points to remember

- LINQ is a language that allows us to query C# objects.
- Entity Framework Core is the most famous and used ORM of the .NET platform.
- The Code First approach states that migration needs to be created after any model changes that involve database tables.
- Combining LINQ and Entity Framework as a wrapper for SQL commands is possible.

Multiple-choice questions

1. **Which alternative represents a false statement for Entity Framework Core?**

    a. It is compatible only with the SQL Server database.

b. The framework allows us to generate the database via C# code.
c. Inner join can be applied in Entity Framework by using the Include method.

2. **Which class needs to be used to create DbSet properties?**
    a. LINQ
    b. System.Collections.Generic.List
    c. DbContext
    d. None
3. **Which method is not supported by the DbSet class?**
    a. Add
    b. Delete
    c. Update
    d. Commit

Answers

1. **a**
2. **c**
3. **d**

Questions

1. Explain what ORM is.
2. Using the code samples of this chapter, create a new product using Entity Framework methods.
3. Create a LINQ statement that returns all the products, starting with the letter “A.”

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 19

Good Practices for .NET Applications

Introduction

Given that you are already familiar with the basic concepts and capabilities of the .NET platform, it is essential to understand good software development practices using this technology to take the maximum benefit of .NET in terms of performance, maintainability, and scalability.

Learning good practices of .NET development will allow you to create robust logging systems based on events, high-performance applications, and excellent reliability in terms of error handling for enterprise applications.

This chapter will cover the essential aspects of logging, dependency injection, tips for application performance, and a more detailed explanation of how to implement good error handling for your .NET applications properly.

Structure

In this chapter, we will discuss good practices applicable to .NET applications, such as the following:

- Dependency injection
- Logging
- Performance
- Exception handling

Objectives

After studying this unit, you should be able to include logging as part of the software development strategy. You will learn how to monitor production environments using logging and how to apply dependency injection. You will also learn to apply performance enhancements for existing .NET

applications and you will understand the principles of error handling in .NET applications

Dependency injection

It is known in software development that if a class has too many dependencies on other classes, it is usually a sign that the project is harder to maintain. It has more potential to have more bugs included over time. A high level of coupling between classes violates some sound principles of software development, such as the Single Responsibility Principle (SRP), and it makes the automation of tests something challenging as it is more difficult to mock external resources or even other classes during the test automation (unit and integration tests).

Dependency Injection minimizes the coupling issue as it is a technique that reduces the dependency between classes. If a class needs to use an object from another class type, the creation of this external object is separated from the class that is consuming it, decoupling usage from creation. This practice increases the reusability of classes and minimizes the number of times a class needs to change over time, possibly replacing dependencies over time without having to change anything in the class that is consuming an object.

The use of Dependency injection can be achieved by correctly using interfaces to pass or transfer objects between classes. If an interface is referenced and not a concrete class, it gives us more flexibility as the classes involved would rely only on the contract established by the interface. Therefore, the primary goal of dependency injection is to remove the high dependency between classes by having a flexible data flow in terms of objects across the system when classes need to reuse objects from other classes.

As an example of dependency for C# classes, imagine you have a scenario where you have a flight ticket booking Website. This Website has an integration with air company services to register the booking. In a simplified scenario, you could have the following Booking Order class, as shown in *Figure 19.1*:

```
BookingOrder.cs
BPBBookChapter19                    BPBBookChapter19.Models.BookingOrder                    Price
1   namespace BPBBookChapter19.Models
2   {
        3 references
3       public class BookingOrder
4       {
            0 references
5           public int CustomerNumber { get; set; }
            0 references
6           public DateTime Date { get; set; }
            0 references
7           public string? AirportOrigin { get; set; }
            0 references
8           public string? AirportDestination { get; set; }
            0 references
9           public decimal Price { get; set; }
10      }
11  }
```

Figure 19.1: *Booking Order class*

Simulating a real scenario for a booking system that has to communicate with external services, it is very common to have a manager class to separate the application's logic from the necessary implementation to establish communication with the external service. Given that the simulation would have the Booking Manager class, as represented in *Figure 19.2:*

```
BookingManager.cs
BPBBookChapter19                    BPBBookChapter19.Models.BookingManager                    SendBooking(BookingOrder bookingOrder)
1   namespace BPBBookChapter19.Models
2   {
        0 references
3       public class BookingManager
4       {
            0 references
5           public async Task<string> SendBooking(BookingOrder bookingOrder)
6           {
7               var bookingSender = new BookingSender();
8               return await bookingSender.SendAsync(bookingOrder);
9           }
10      }
11  }
12
```

Figure 19.2: *Booking Manager class*

In this example, the Booking Manager class has a method called "**`SendBooking`**," which refers to the BookingSender class responsible for sending the request to the external air company service. The implementation

of the Booking Sender class looks like *Figure 19.3* for demonstration purposes:

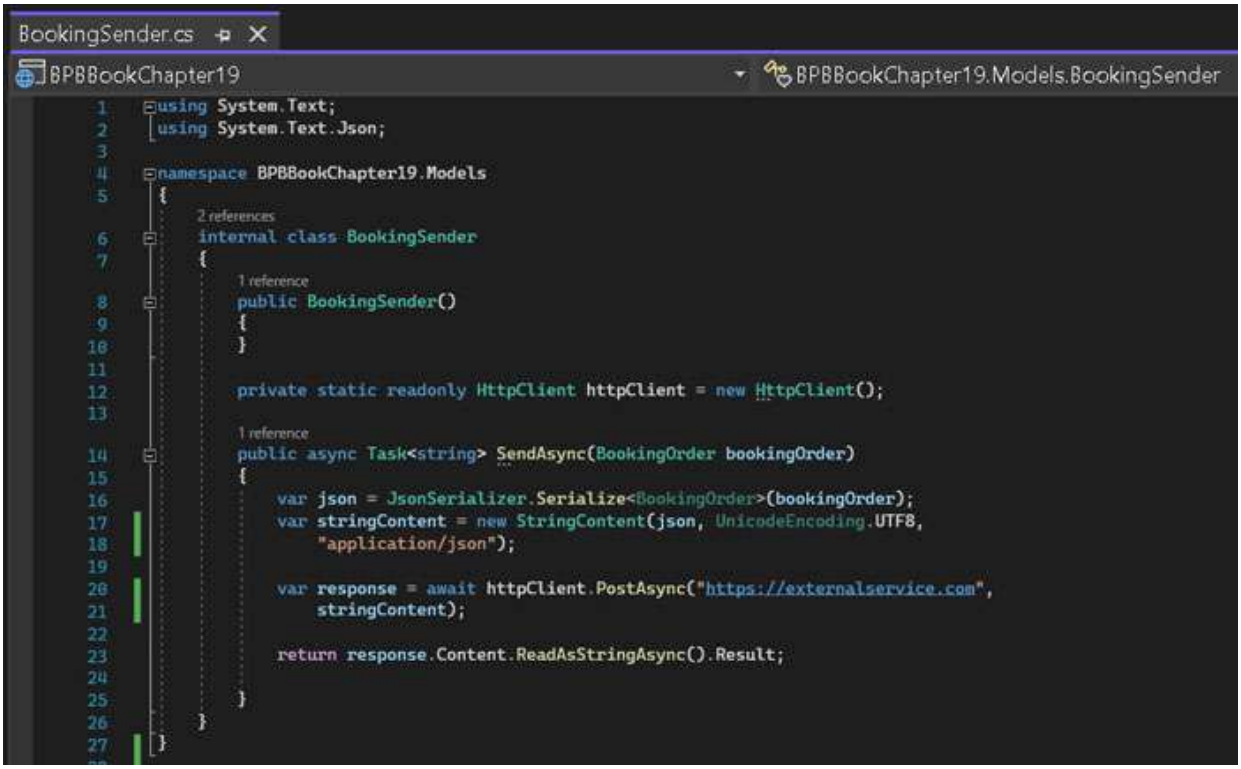


Figure 19.3: *Booking sender class*

The Booking Sender class receives a Booking Order object, serializes the order to a JSON object, and sends it to the external air company service via an HTTP request. The endpoint on line 20 is fictional just for studying purposes.

Given the three classes demonstrated so far, there is an explicit dependency between the **`BookingManager`** and the **`BookingSender`** classes, making the maintainability of these classes compromised as any extension or change to the requirements would directly affect both classes. Imagine you have a scenario or a new request to introduce a new air company service that is not handled by the **`BookingSender`** class. In that case, you would need to create a new **`BookingManager`** class to refer to the new type of air company service. This coupling increases complexity and makes the code not reusable or safe from interference over time. This is precisely the problem that Dependency Injection is meant to solve.

The .NET platform contains a built-in feature that simplifies the implementation and use of Dependency Injection by having the concept of

an IoC Container. This framework implements automatic dependency injection, managing the lifecycle of objects and abstracting the injection of classes across the application. The IoC Container contains the following features:

- **Registration** is the definition of the types of objects that the application must create and inject in all the necessary instances.
- **Resolution** is the actual injection of objects via constructor automatically across the application
- **Disposition** is managing the lifetime of the dependencies being injected across the application.

To refactor our Booking Service example to use dependency injection, the first that needs to be done is to abstract the Booking classes to implement interfaces. Therefore, the **`BookingSender`** class needs to have an interface, as shown in *Figure 19.4*:

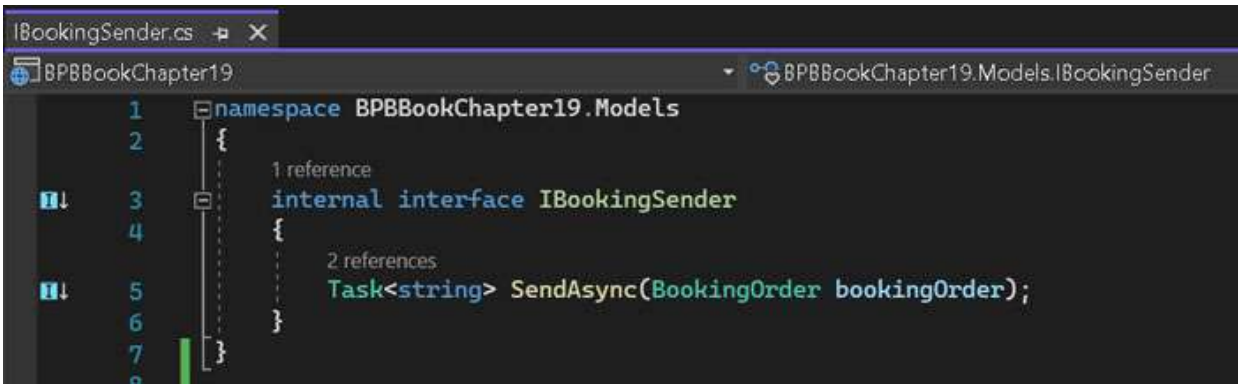


Figure 19.4: *Booking Sender interface*

As the idea is to abstract all the dependencies, the Booking Manager class must have an interface as well, as demonstrated in *Figure 19.5*:

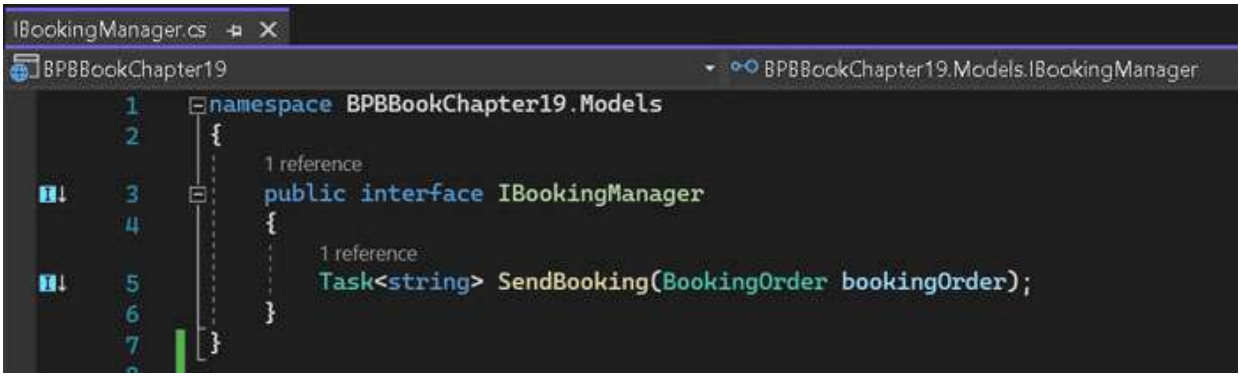


Figure 19.5: *Booking Manager interface*

As we already have the underlying interfaces for the Booking classes, the next step is to force the concrete classes to implement those. Therefore, change the **`BookingSender`** class to implement the related interface, as highlighted in *Figure 19.6*:

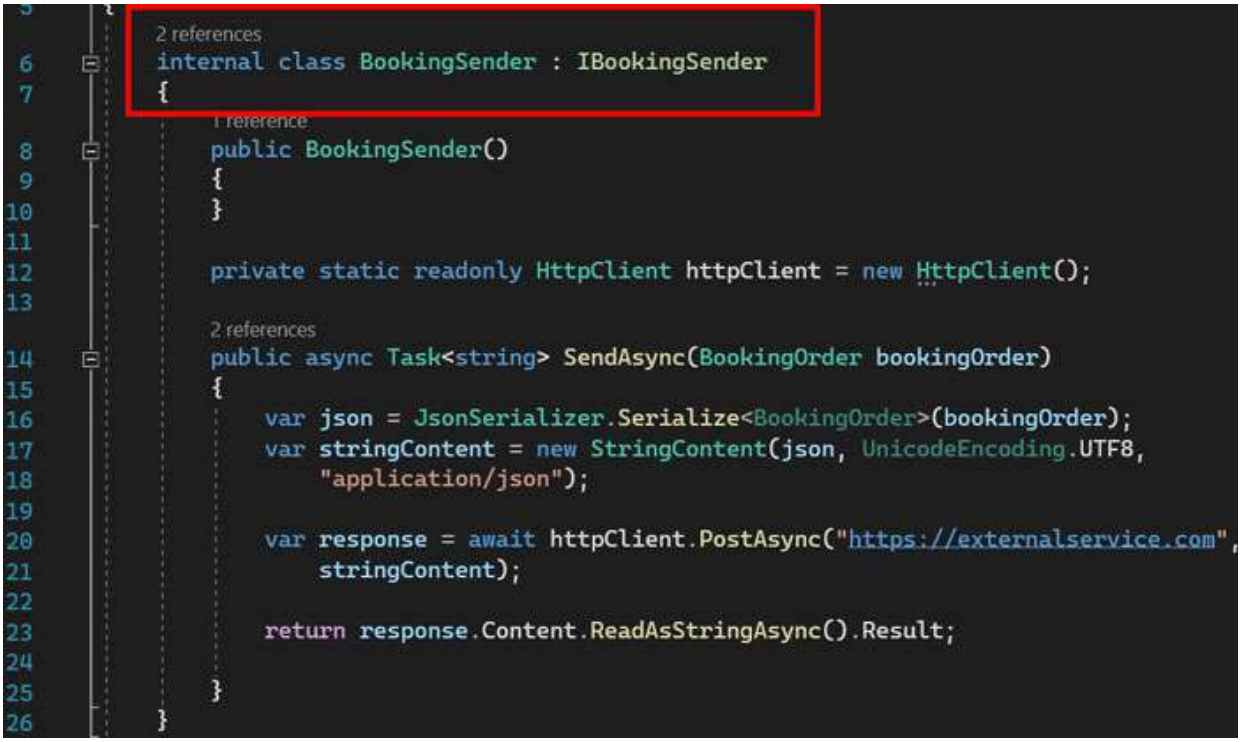


Figure 19.6: *Booking Sender interface implementation*

The same thing needs to be done for the Booking Manager class, forcing the implementation of the new interface, as shown in *Figure 19.7*:

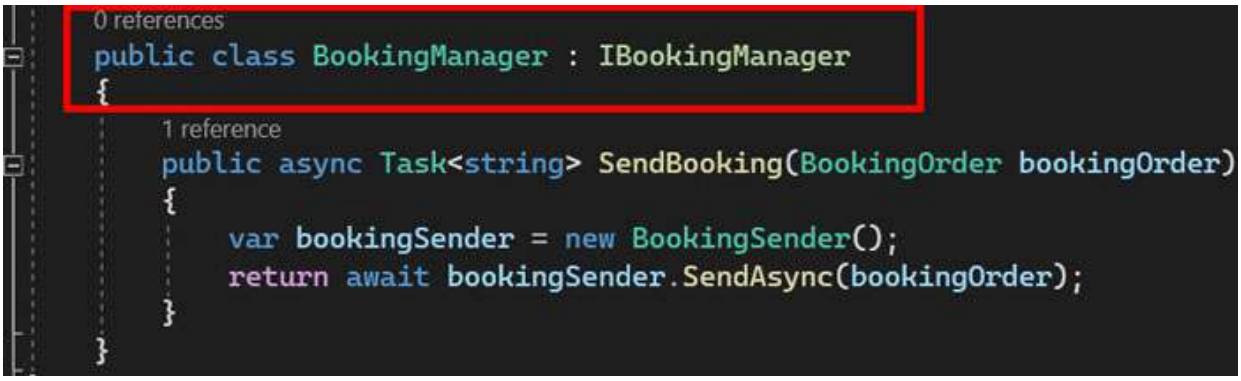


Figure 19.7: *Booking Manager interface implementation*

The Booking classes already implemented the related interfaces; however, this does not reduce the dependency between the two classes per se as the Booking Manager class still directly refers to the concrete Booking Sender class. To solve this issue, the Booking Manager class needs to receive the Booking Sender object in the constructor, mandatorily, but referring to the interface and not the concrete class, as shown in *Figure 19.8*:

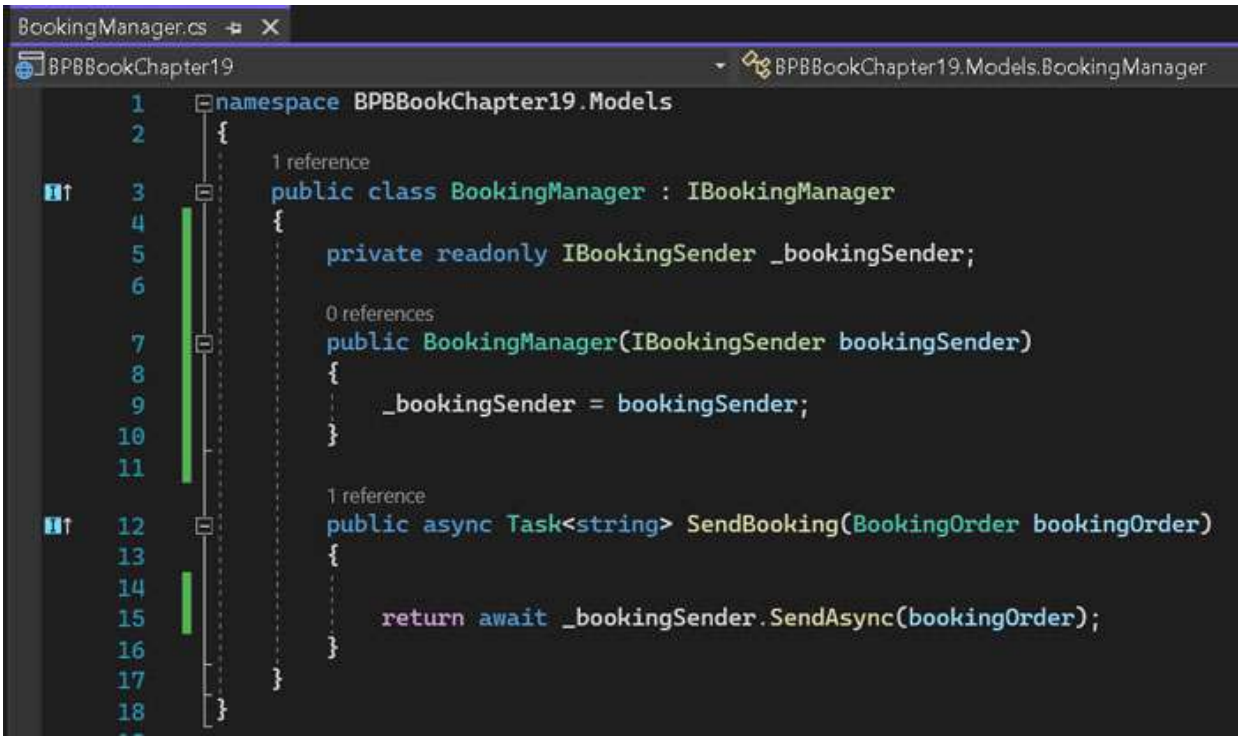


Figure 19.8: *Dependency injection for Booking Manager*

As seen in *Figure 19.8*, the Booking Manager class no longer creates an instance of the Booking Sender class, with the dependency being handled by the class constructor. With these simple changes, there is no direct dependency between the two classes, being possible to change the Booking Sender class without directly affecting the Booking Manager class.

As the dependency between the two classes was already resolved, the next step is configuring the application to use the default Dependency Injection feature from .NET. For this chapter, an Asp.Net Core Web API project was created, targeting to .NET 6 version to demonstrate the registration of injection of dependencies. In this case, you need to configure it on the **`Program.cs`** file, as highlighted in *Figure 19.9*:

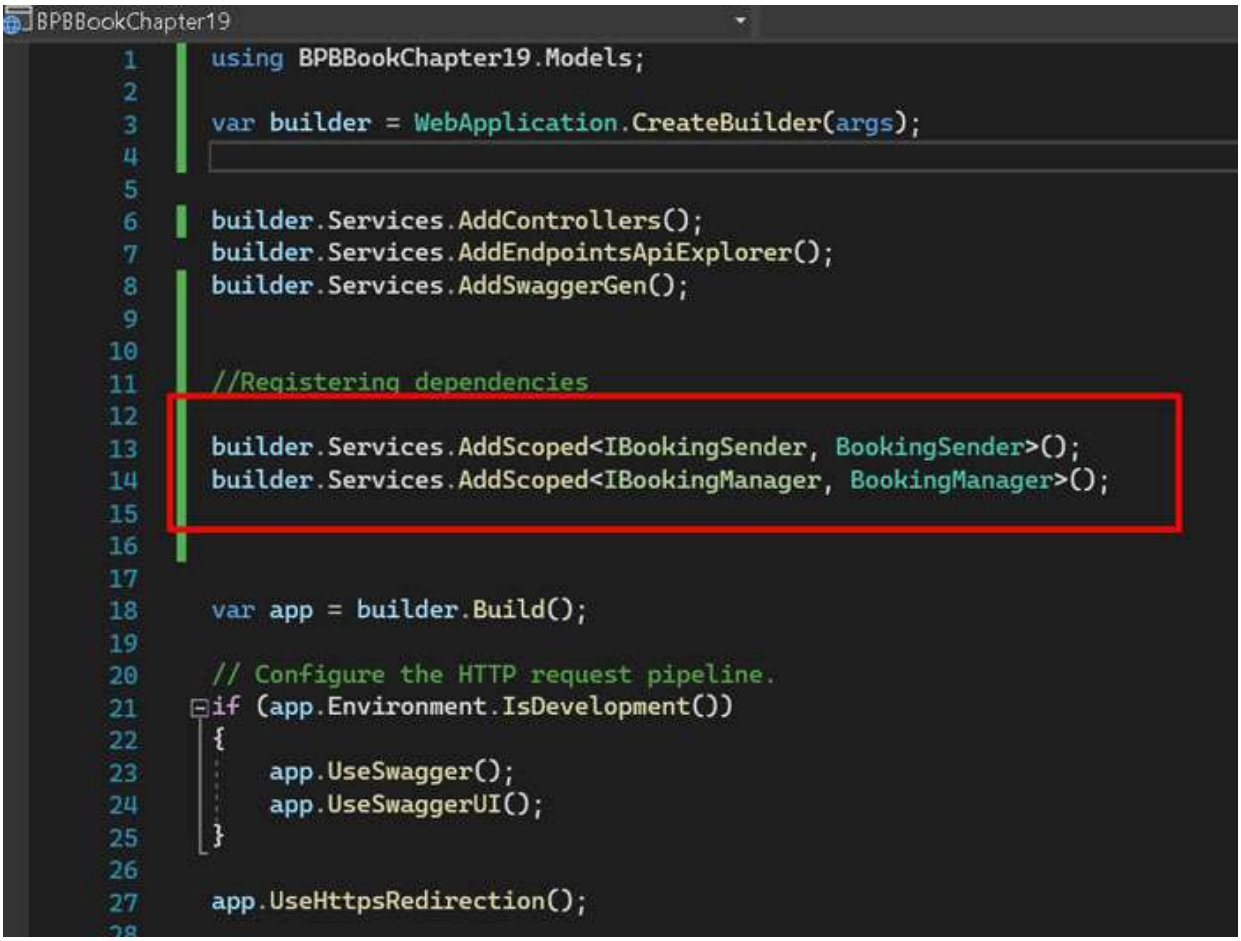


Figure 19.9: *Registration of dependencies*

Note that the **`AddScoped`** method was used to determine the lifetime of the dependency. In this case, it will follow the lifetime of a request in the Web API. There is a possibility of using the **`AddSingleton`** method to determine that all the requests will get the same object instance. You have to use what is more suitable based on functional and non-functional requirements for your application. The last thing that needs to be done in the context of the Web API is to refer to the Booking Manager interface at the Controller level, as shown in *Figure 19.10*:

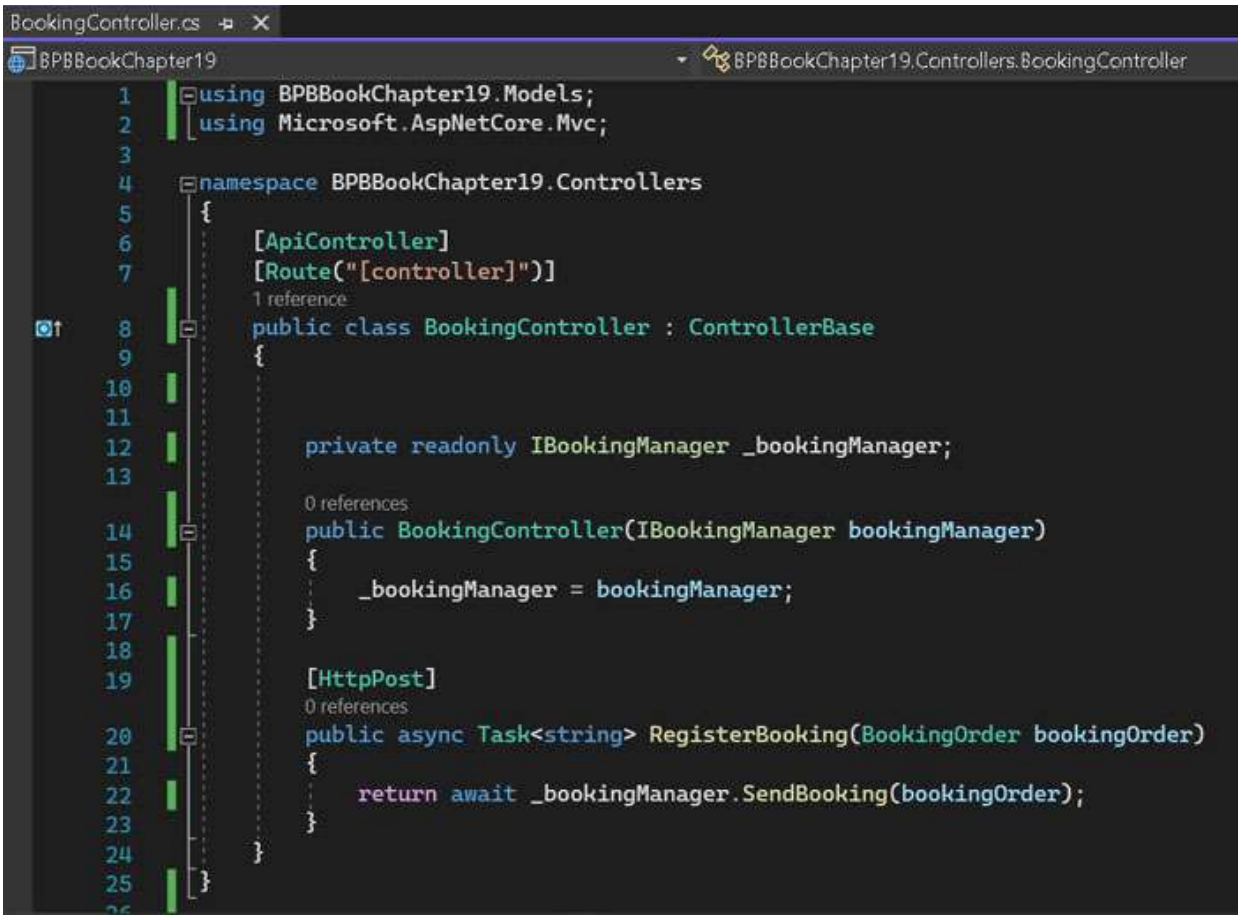


Figure 19.10: *Booking Controller*

This Booking Controller receives in the class constructor a Booking Manager object from any class that implements the **`IBookingManager`** interface as we registered in the **`Program.cs`** file the Dependency Injection of the same interface. Every time an instance of this Controller is created, the IoC Container automatically passes a Booking Manager object to the Controller. The same happens with the injection of the Booking Sender class into the Booking Manager class. There is no need to pass the instance of these classes programmatically, as the .NET applications already take care of that based on the configuration of the **`Program.cs`** file.

Logging

Logging is one of the most important non-functional requirements in any software project. It helps to figure out problems in production environments and gives us useful information on how the application runs. Implementing a

good logging system will provide valuable data for troubleshooting when an error happens in production, reducing the time and costs of the investigation.

The native logging option for the .NET application uses the ILogger interface from Dependency Injection. This is added by default in various types of projects in the .NET platform, including Web APIs, as you can see highlighted in *Figure 19.11*:

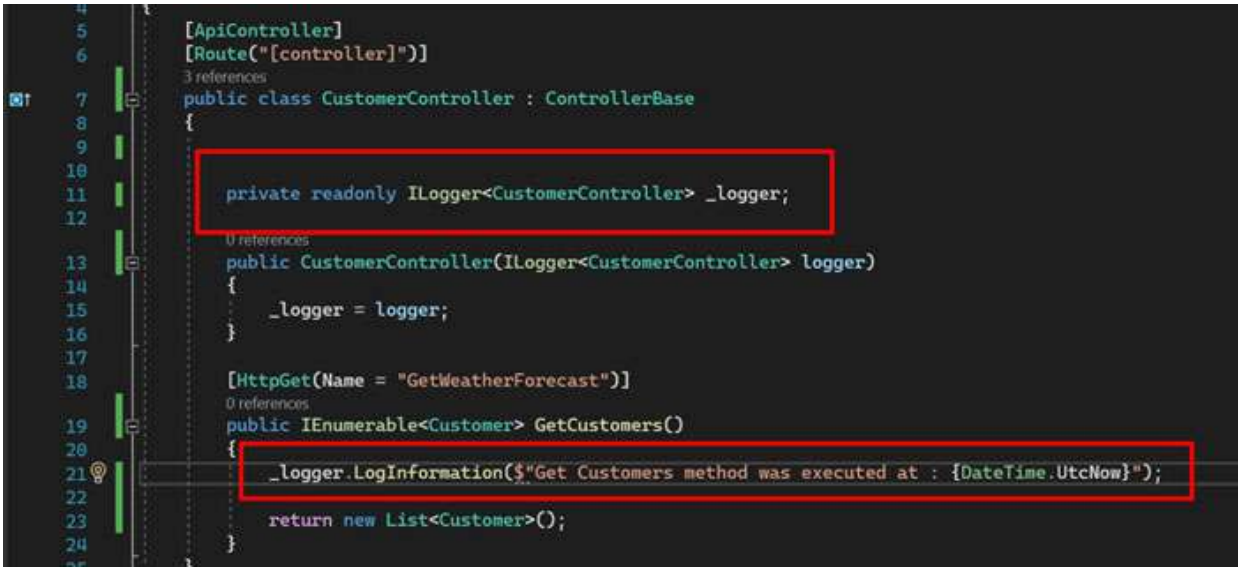


Figure 19.11: *Logging interface*

A more general setting for the logs in the application can be placed in the app settings file, determining the level of logging that is desired for the application in general, as shown in *Figure 19.12*:

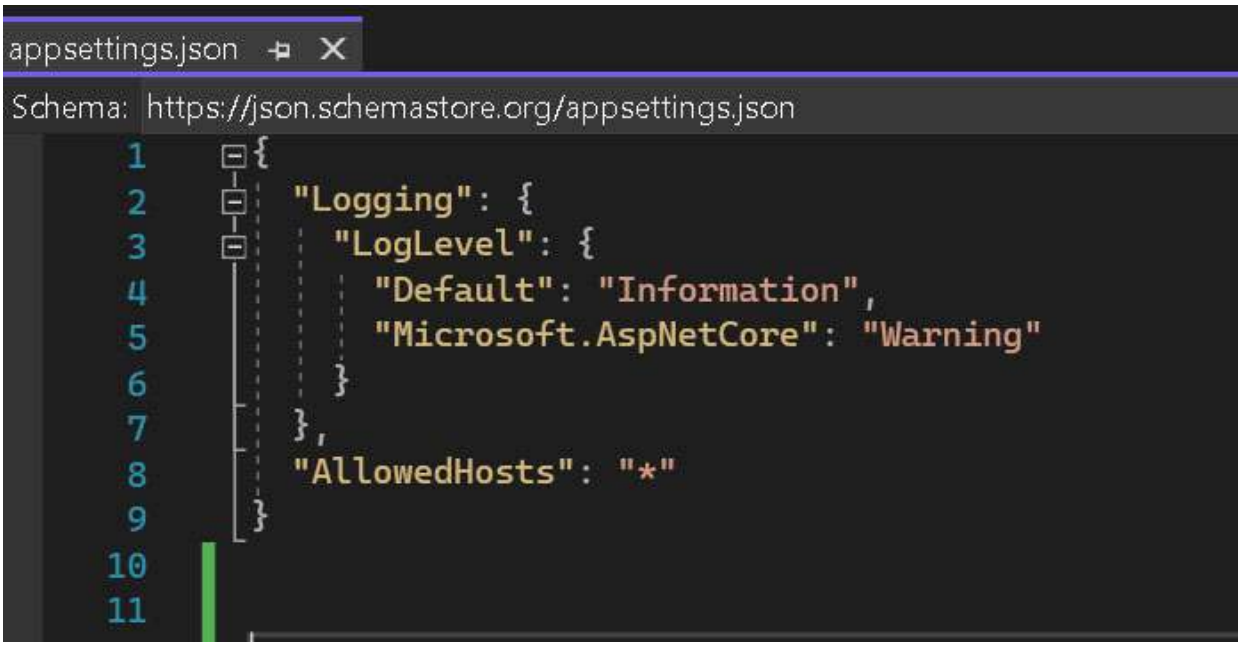


Figure 19.12: *Logging configuration*

The **`LogLevel`** property determines the minimum level that is applicable for the categories associated with the log and its servers to indicate the severity of the log, having the following options to use:

- Trace
- Debug
- Information
- Warning
- Error
- Critical
- None

Based on the **`LogLevel`** defined, this feature is enabled in the application respecting this as the minimum level used, being possible to specify different values for this property for Debug or Release modes.

Performance

Good application performance depends on various factors, such as infrastructure, programming language, software architecture, workload, database, and many other components. Despite some common factors that

would apply to any technology for software development, the .NET platform has good practices that contribute to a better performance of enterprise applications.

One of these practices is the correct use of asynchronous methods across the application, including Controllers for Asp.Net Core applications, Entity Framework Core operations, and much more. If the async keyword is not used in any method during a request, the server's main thread is locked, decreasing the performance for new requests to the server. As operating systems and modern servers support multiple cores for CPUs, using multiple threads by the application is crucial for performance in general.

As seen in *Figure 19.13*, once a method is marked as async, the await operator needs to be used in combination when another async method is called in the tree of events:

Figure 19.13: *Async method*

Some native implementations within the .NET platform already use async methods by default to maximize performance. One good example of that is the manipulation of Stream objects. Since .NET Core, it is mandatory to parse and manipulate Streams asynchronously.

Another critical point regarding performance for .NET applications is the correct use of HTTP client objects. Even though HttpClient has an implementation of an **`IDisposable`** interface to clean the memory after its use, instances of **`HttpClient`** may leave socket connections open in the server for longer than desired. Each server has a limitation on the number of socket connections available; therefore, the lifetime for HTTP Client connections is essential for the server's performance. To remediate this problem, you can use an **`HttpClientFactory`** object via Dependency Injection to reuse the same HttpClient across the application to multiple requests, being the Singleton Scope suitable for most cases when the **`HttpClientFactory`** is registered on the services for Dependency Injection.

Exception handling

One of the most important aspects of any application is the level of reliability to final users or any consumer of the application. Preventing the app from crashing is not a trivial task in software development. The solution is related to how we implement Exception Handling in the application.

Therefore, use try/catch blocks in all the parts of the application that can cause errors and use custom exceptions to determine how the application will behave for each type of exception that may occur during the execution of the application. In this case, a combination of Exception Handling and a good Logging system is crucial to catch all the needed information to improve the software over time and actively investigate issues in Production environments.

Conclusion

In this chapter, you had the opportunity to learn basic concepts to apply general good practices for .NET applications, including logging, exception handling, dependency injection, and performance enhancements.

In the next chapter, you will learn the most common architecture patterns used for enterprise .NET applications.

Points to remember

- Dependency Injection uses the default IoC Container feature in .NET applications.
- It is possible to determine the lifetime of a dependency when it is registered in the application services.
- It is recommended to use logging from the beginning of any project.
- The correct use of async methods is one of the essential practices that helps any project to have a good performance.

Multiple-choice questions

1. **Which alternative represents a false log level for .NET applications?**

a. Trace
b. Warning
c. Release
d. Critical
e. Debug

2. **Which interface is used by default for logging in to the .NET platform?**

a. Application Insights
b. ILog
c. None
d. ILogging

Answers

1. **c**
2. **d**

Questions

1. Explain what Dependency Injection is.
2. Using the code samples of this chapter, create a new implementation of the IBookingSender class, injecting the class via Dependency Injection.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 20

Architecture Concepts for .NET Applications

Introduction

Developing a robust application is challenging as it involves definitions of software architecture, including particular decisions around the application's platform, such as web servers and mobile or desktop devices. The .NET platform has established good practices for software architecture decisions over the years.

Learning good software architecture practices for .NET applications will allow you to properly understand the platform's capabilities. It will also allow you to build robust and scalable enterprise applications with a higher probability of success.

This chapter will cover the essential aspects of software architecture for .NET applications, including good practices around Cloud apps, Asp.Net Core applications, and DevOps, showing options to modernize existing web applications.

Structure

In this chapter, we will discuss the following good software architecture practices applicable to .NET applications:

- Architecture practices for web applications
- Cloud applications
- Concepts of DevOps
- Introduction to microservices in the .NET platform
- Design pattern concepts

Objectives

After studying this unit, you should be able to modernize existing web applications to be compatible with cloud infrastructure. You will have a better understanding of resources on Azure for .NET applications. Also, you will understand the basic principles of microservice architecture involving the .NET platform.

Architecture practices for web applications

A good architecture for any web application involves aspects of scalability and costs regarding infrastructure directly and hosting in general. The application must be kept simple enough to support the business needs, facilitate maintainability, and provide modern deployment methods using automation tools. Moving legacy applications to a cloud infrastructure greatly benefits achieving these goals. It does not necessarily represent a significant need for re-design or re-architecture of the application itself.

Migration to the cloud

There are many strategies for the migration of application infrastructure to the cloud. In many cases, it might be convenient to replicate on the cloud a similar infrastructure that an application may run against in on-premise environments. It can be easily achieved by creating virtual machines of similar configuration on Azure. This approach may be an excellent first step in cloud migration, as it does not involve changes to the application itself. Still, the application will be hosted only with a change on the servers.

However, in the cases where a more modern and scalable approach is required, there is a possibility of hosting the web application on the Azure App Service, which follows the concept of **Infrastructure as a Service** (**IaaS**), facilitating the management of cloud infrastructure by taking the benefits of auto-scaling configuration presented on the Azure App Service. This model may involve some level of coding changes to the application to be compatible with this type of cloud infrastructure. It is an excellent alternative when the application can be deployed in Docker Containers.

A more complex migration to the cloud would involve the re-architecture of the application to use the concept of Cloud-Native applications, which

involves the use of serverless microservices (Azure Functions), Azure Kubernetes, and other cloud resources that require a total change of the application architecture to be compatible with this scheme.

Azure App Service is the most common hosting service available on the Microsoft cloud platform to host web applications and jobs. In the context of Asp.Net Core applications, this cloud resource is compatible with the possibility of mixing app configurations with infrastructure settings. In terms of architecture decisions for Asp.Net Core applications in general, migrating to the cloud may represent an excellent first step to modernizing your web applications and taking advantage of scalability on the infrastructure side.

Single Page Applications (SPAs)

In terms of web development for modern applications, there are two basic ways to structure web applications in general: running the entire application on the server side or splitting the responsibility of the execution between the server and client (browser). There are pros and cons to each approach. Still, in terms of performance, the use of Single Page Application architecture has proven to be a perfect alternative as it allows the delegation of a massive amount of processes to be executed on the client side, giving the users a better experience and reducing costs of server infrastructure for more complex projects.

Architecture decisions must support non-functional requirements, such as security, performance, scalability, and other factors. Therefore, choosing an approach by running the application on the server side or mixing the process between the client and server depends directly on the nature of the application, such as:

- The complexity of the user interface
- Familiarity with the team with frameworks that support Single Page Applications
- Security concerns in terms of the use of JavaScript by the web application in the client
- Level of compatibility across browsers if the application is using complex logic to manipulate the user interface

Suppose the application needs to be executed entirely on the server, including the user interface. In that case, using the traditional Asp.Net MVC project template might be helpful as it has been an established project type over the last ten years in the market. Another good option, in this case, would be using Razor pages, which runs all the processes exclusively on the server and has a lighter template.

On the other hand, if the web application is suitable for the Single Page Application architecture, there is a possibility of combining Asp.Net Core Web API projects in the backend with traditional JavaScript frameworks for SPAs, such as Angular, React, and VueJS. The Visual Studio IDE contains templates for Angular and React applications, and technical communities and individual contributors become available templates for VueJS.

Suppose your team does not have familiarity with JavaScript and TypeScript. In that case, Blazor WebAssembly represents a great alternative as this type of project allows C# developers to build Single Page Applications using C# language instead of JavaScript. Blazor represents a considerable innovation in the .NET platform as it allows the creation of Razor components that can be shared across other Asp.Net Core applications. It has a high performance in the browser as it uses WebAssembly to execute C# language directly on the client side.

Cloud applications

As demonstrated in this book, .NET applications have high integration capabilities with Azure and other Microsoft services as all these features and resources are kept by the same company, following a standard process to build SDKs and libraries. The definition of an exemplary architecture plan for .NET applications involves decisions around the infrastructure where the applications will be hosted, and cloud infrastructure became a viable option for most cases.

Azure App Service and containers

It is possible to host web applications using Azure App Services, an **Infrastructure As a Service** (**IaaS**) resource, keeping control of certain aspects of the environment and configuration using containers. You can use Docker images and deploy containers with pre-configured applications,

with automatic capabilities for scalability without significant setup effort. The use of containers allows you to transition between different cloud providers without making any changes in the application, representing a considerable advantage in terms of costs and strategic planning for software projects in general.

Serverless compute

The Azure platform allows the creation of serverless functions using C#, NodeJS, Python, and Java, creating a model for scalability. Azure functions is an excellent technology for running code, integrating with other Azure resources, and managing data. Serverless architecture is suitable when there is a need for automatic scaling and the functions' routines are used for multiple projects.

Durable functions are a great alternative within Azure for more complex tasks, as it allows long-running tasks by orchestrating multiple Azure functions following a specified workflow.

Azure Kubernetes Service (AKS)

When a more complex architecture for .NET applications based on a microservices approach needs to occur, using a container orchestrator is vital to guarantee scalability without overhead in infrastructure management. **Azure supports Kubernetes** (**AKS**) which is one of the most popular tools for orchestrating containers.

Kubernetes is an open-source technology that facilitates deploying and managing thousands of containers for large applications. It is compatible with the idea of cloud-native applications, which consists in building applications that would be compatible with all major cloud providers, such as Microsoft, Google, and Amazon. Therefore, if your application runs using containers and Kubernetes on Azure, it can be easily migrated to other cloud providers without making significant deployment changes.

DevOps

Software development drastically changed over the last decade due to the high complexity of the web, desktop, and mobile applications needed to

meet the demand of massive global users. The frequency of releases in the **Software As a Service** (**SaaS**) model increased to have production deployment multiple times a day without the downtime of the related systems. This new model represents a considerable challenge regarding Engineering processes and Release Management. It requires maturity in automation and a clear understanding of software development techniques that allow multiple deployments daily without compromising user experience for critical systems.

In the context of this more complex environment for software development, deployment, and release management, a well-established DevOps process in any organization is critical to the success of major projects, as it is strictly related to a high capacity of correction of production issues while keeping a high performance of multiple applications, avoiding business risks.

A good DevOps process involves an intense collaboration between engineers, infrastructure teams, and the whole software organization. It also involves using automation for tests, deployment, and release management. Therefore, .NET applications must have an architecture that supports these non-functional requirements regarding DevOps, which directly involves the use of techniques to facilitate automation of a test running suit and automatic deployment.

Using containerized applications is a good practice in terms of facilitating deployment to different environments and avoiding extra costs and effort related to infrastructure configuration. Consider using Docker containers for your .NET applications as part of the early technical decisions for the projects. For systems-based cloud architecture, it is essential to consider the possibility of failure for any component part of the solution. For instance, one of the services responsible for specific operations might fail due to connectivity or other issues. In this case, the infrastructure architecture must be resilient to provide alternatives in failures, such as auto-scaling, redirection to another service, retry policies, and much more. Managing a complex transaction for a microservices architecture is not a trivial task. Many problems must be anticipated as part of the system design to support a reliable and resilient solution for mission-critical applications.

Introduction to microservices

Microservices architecture is a complex topic that has been in discussion for over a decade in the market, with debates around the best scenarios to use. This book does not have the purpose of explaining this in detail architecture. Still, it only intends to highlight the essential points to support you in a general decision-making process around software architecture for .NET applications.

This architectural model allows you to combine multiple specialized services, independent from each other, organized by business requirements with a loosely coupled approach between different applications. Imagine you have a complex enterprise system to build, like an intranet for a company with thousands of employees and many departments, such as Human Resources, Finances, Sales, Recruitment, Engineering, IT Operations, Quality Assurance, and much more. In this context, you can create a single colossal application that will meet the needs of thousands of users from different departments. This software approach is usually called the monolith model.

In this context of having a single application, if a change is required in the module used by one department, any release of the software may impact the entire organization, as the multiple modules in the system share the same infrastructure, which includes the application server, database, storage, and other resources. Depending on how often an application needs to be deployed to production environments, having multiple downtimes a day for the entire company might represent a risk to the business and the software engineering team's reputation.

To avoid these risks and adverse effects of keeping large systems as a single extensive application for all departments in the company, it is possible to split the same application into smaller ones, independent from each other, which means that different applications would be hosted in different servers. They would also have distinct databases, dedicated infrastructure management in scale, backup policies, security, and much more. This may represent a considerable challenge in coordination and management and sometimes may not be cost-effective. However, it is a good approach when a company is looking for an independently deployable capability for different components in the system.

Given the nature of microservices architecture, the key technologies and approaches that .NET developers can take to address complex problems

with this multi-application model. The most important one is the correct use of Docker containers, which provides a cost-effective solution as it reduces issues in deployments and effort in infrastructure configuration. Docker containers became highly supported by all the big cloud providers in the market, allowing an easy cross-platform transition.

Microservices have become the standard distributed system approach over the last few years. A good strategy of cloud resources management is also a key factor in the success of complex projects representing critical business applications.

Design pattern concepts

Correctly using any Design Pattern in real projects requires a strong background in object-oriented programming paradigm knowledge. Also, it requires a deep comprehension of all the technical and business requirements behind the project being built. The Design patterns used in the market were defined to solve specific problems in software development. Before using any of those, it is necessary to identify if the project has a similar problem.

Some of the design patterns that are broadly used in the market use many concepts related to the object-oriented programming paradigm, such as interface, inheritance, polymorphism, abstraction, protection modifiers, and other important concepts that directly affect the development of components and a whole software project. Therefore, getting familiar with all the object-oriented paradigms and SOLID principles is extremely important.

Software development is a complex process that involves applying many distinct techniques; being essential to get yourself familiar with the most basic concepts of C# language until the use of advanced features becomes natural daily. Most good practices in software development are common sense among developers, including for projects built using the .NET platform and C# language. Therefore, some good practices, such as the SOLID principles, will be used in real projects. Still, they will not be explicitly mentioned, and their identification will not be so evident in many cases.

Simplicity must be the core of any project regarding technical aspects, and the code should be readable and easy to maintain. Therefore, this should be considered in the technical decisions when the project architecture is being defined in a planning process. Design patterns and any other technique represent solutions for specific problems and should be carefully used to not include extra complexity in projects that may not require using them.

In specific scenarios, if the development team is not familiar with Design patterns, in case of a project requires to use of a specific one, it is recommended to ensure that all the developers in the team know the basic concepts of the object-oriented paradigm, SOLID principles, and the design pattern that needs to be applied in the project. Suppose the team is not familiar with them. In that case, it is highly recommended to start a training process to have all the developers on the same page, to work on the new features, and follow all the recommended approaches when a design pattern is used.

The complexity of software development increased exponentially in the last few years because of all the requirements behind global projects, such as social media, streaming platforms, online banks, and mobile entertainment applications. All these kinds of applications have something in common: they face considerable challenges in terms of scalability, being cloud-based solutions, which means that the complexity of these projects is not only related to programming language aspects but also in terms of software architecture and infrastructure. Keeping any application as simple as possible regarding coding and technical aspects is always a good approach. A single project may have any extra concerns during the development process, and simplicity, readability, and the correct observance of the SOLID principles might represent the key to technical success for projects in general.

Conclusion

In this chapter, you learned the basic architecture practices that are part of the normal development of .NET applications, including microservices architecture, containers, Kubernetes, Single Page Applications, and considerations regarding Design patterns.

In the next chapter, you will be able to apply all the concepts learned in the book by developing an enterprise application following a hands-on

approach.

Points to remember

- Microservices need to consider separate infrastructure for each application.
- The use of design patterns requires maturity in the development team on knowledge of object-oriented paradigms and SOLID principles.
- Docker containers became the standard approach for agnostic deployment across multiple cloud providers.
- The .NET platform provides its Single Page Application (Blazor) based on WebAssembly, which is an excellent alternative if the development team is unfamiliar with JavaScript.

Questions

1. Explain in which scenarios the use of microservices architecture might be suitable.
2. Based on your current knowledge of .NET and C# language, create an Asp.Net Core application that supports the Docker container.
3. Why is DevOps important?
4. List which resources on Azure can be considered serverless computing features.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 21

Creating an Enterprise Application in .NET

Introduction

Building an enterprise application using the most recent .NET version and following good software development practices is the best way to gain experience and understand the platform's full capabilities.

In this chapter, you will get the opportunity to practice everything you have learned since you started the journey from the beginning of this book. This step-by-step approach will allow you to build an application from scratch using Entity Framework, Blazor, Tests, and UI Frameworks.

Structure

In this chapter, you will get the opportunity to practice the following concepts:

- Application requirements
- Creation of the application
- Creation of domain models
- Entity Framework Configuration
- Creation of a Business Logic layer
- Creation of Front-end

Objectives

After studying this unit, you should be able to create simple enterprise applications that involve front-end, back-end, and interactions with the database using Entity Framework. Additionally, you will be capable of

structuring a suite of tests and separating the application into layers to facilitate maintainability.

Application requirements

The main purpose of this chapter is to give you hands-on practice to experience the most basic concepts mentioned and explained in this book, including object-oriented programming paradigm, SOLID principles, Entity Framework, LINQ, Blazor WebAssembly, and Blazor Server. This extensive experience requires developing an enterprise application that is complex enough to involve multiple classes, business logic, and a precise scenario in terms of requirements.

Given that this chapter covers the development of an enterprise application that meets the basic requirements for a logistic company responsible for controlling information on container transport via maritime modals. This involves the creation of the following entities:

- **Container type**: Identifies a single container by its name.
- **Country**: Registration of countries, including name and initials.
- **Customer**: Creating customers via application, including name, address, country, and notes associated with each record.
- **Port**: Registration of ports, including name for identification and country.
- **Booking**: The primary model of the application, which consists of an embark date, arrival date, container type, and customer associated with the booking.

Considering these five entities, the application must follow a similar structure regarding classes, business logic, and controllers for the Web application. Each model must have its corresponding database table, and the entire data has to be managed by Entity Framework.

In terms of application architecture, it must be a Single Page Application using Blazor Web Assembly, with the front-end making requests to the back end via Web API requests. Each controller must be hosted in a Blazor Server application, and all the application models must be shared across the front and back end without having to rewrite any of them twice.

Creating the application

Using the most recent Visual Studio 2022 version, create a new Blazor WebAssembly application using the default template available, giving any desired name, as shown in *Figure 21.1*:

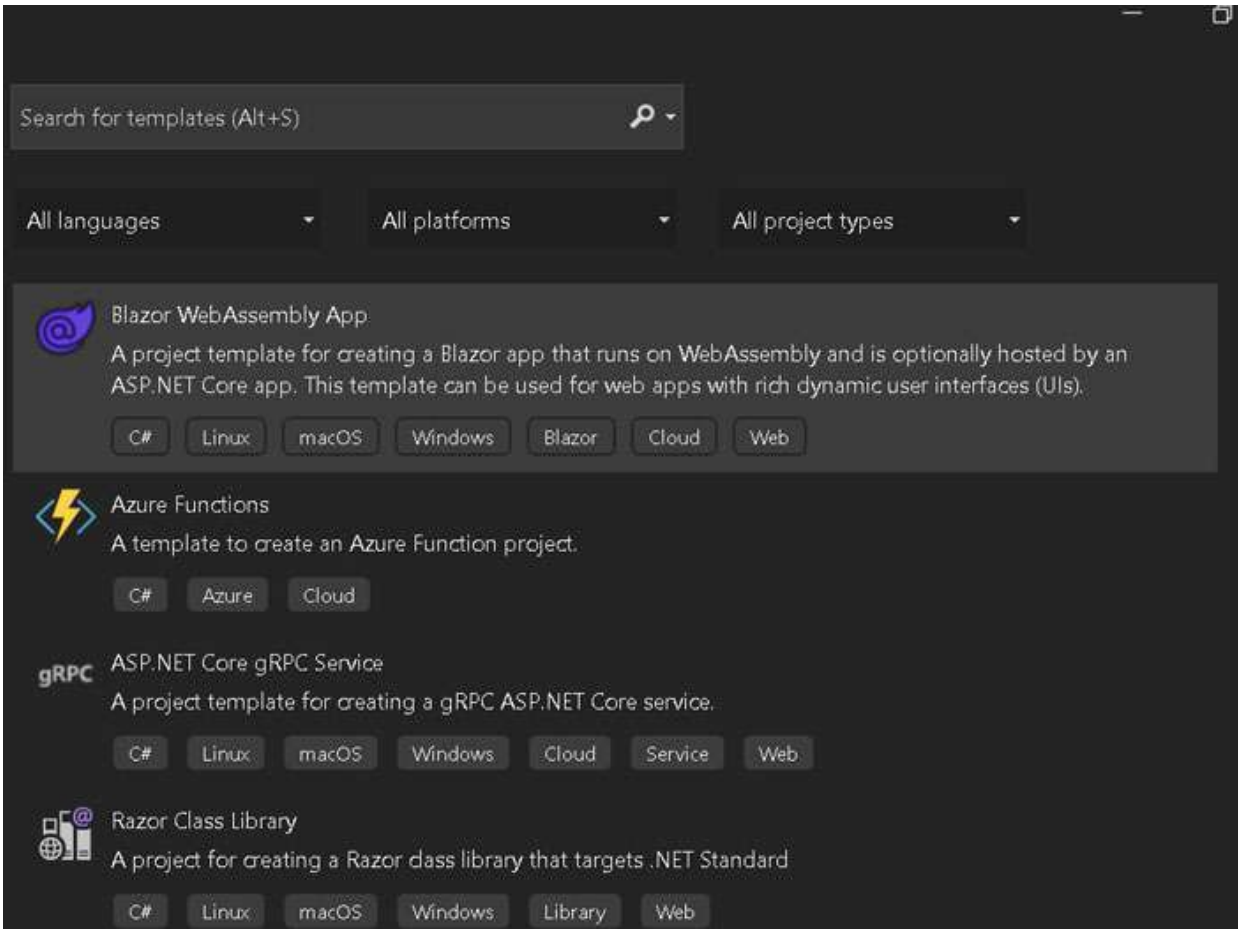


Figure 21.1: *Blazor WebAssembly*

Make sure you choose the Blazor WebAssembly project template. After this, give a project a name. In this example, the name of the application is BookingService, as shown in *Figure 21.2*:

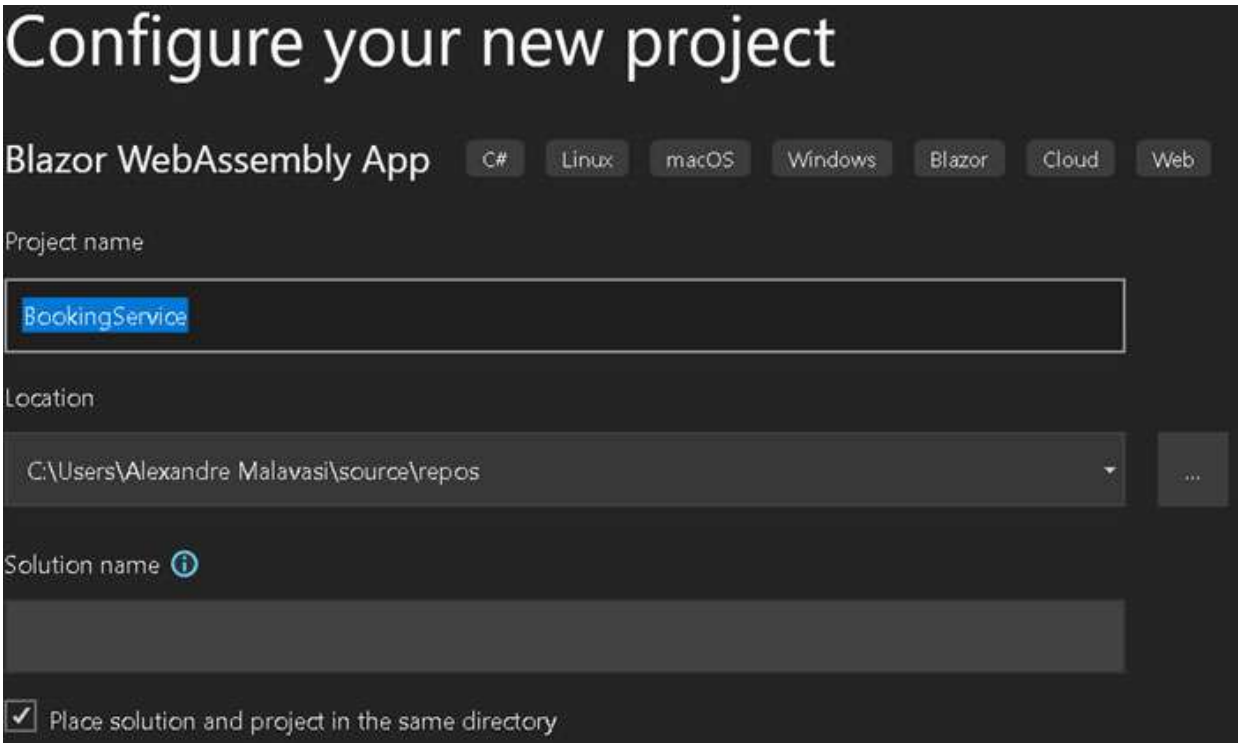


Figure 21.2: *Project name*

`Blazor WebAssembly` projects do not have back-end implementation by default unless the Asp.Net Core Hosted version is chosen along with the front-end application. This hosted version creates three projects in the Solution: Blazor WebAssembly, Blazor Server, and Shared projects. The **`Asp.Net Hosted`** option looks like the representation in *Figure 21.3*:

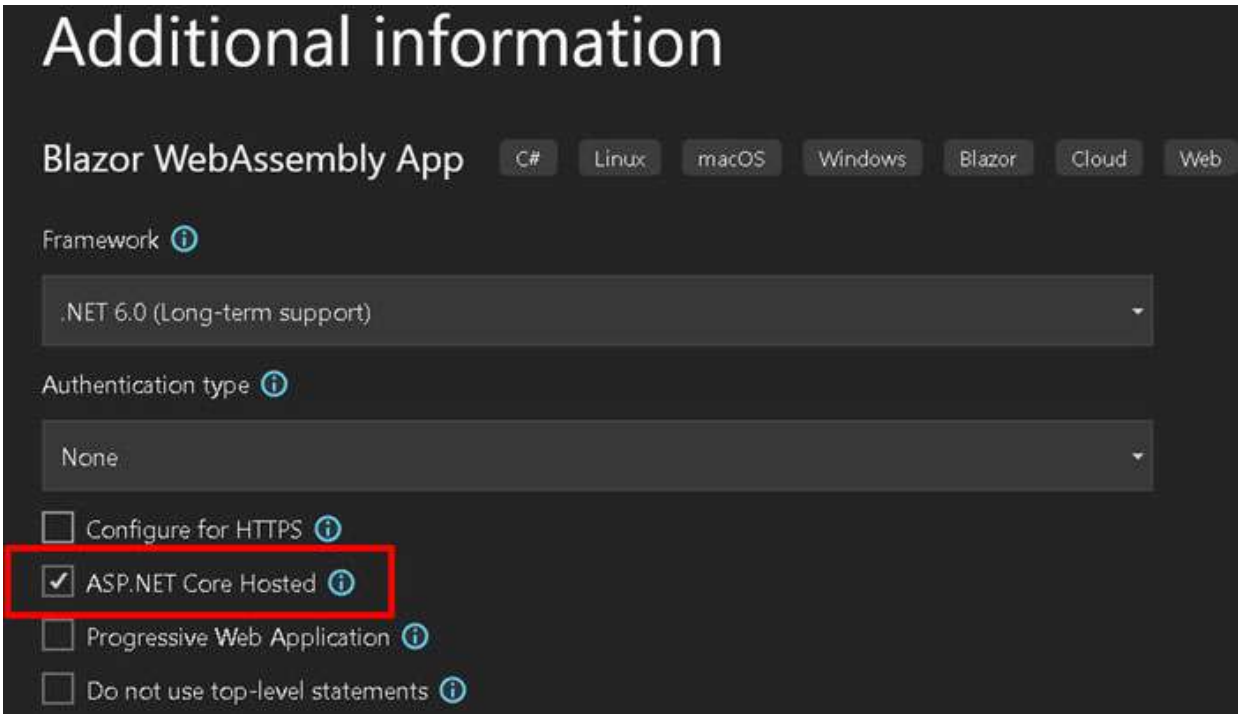


Figure 21.3: *Asp.Net Hosted*

After confirming the creation of the application, the Visual Studio will show a Solution that contains three projects, as shown in *Figure 21.4*:

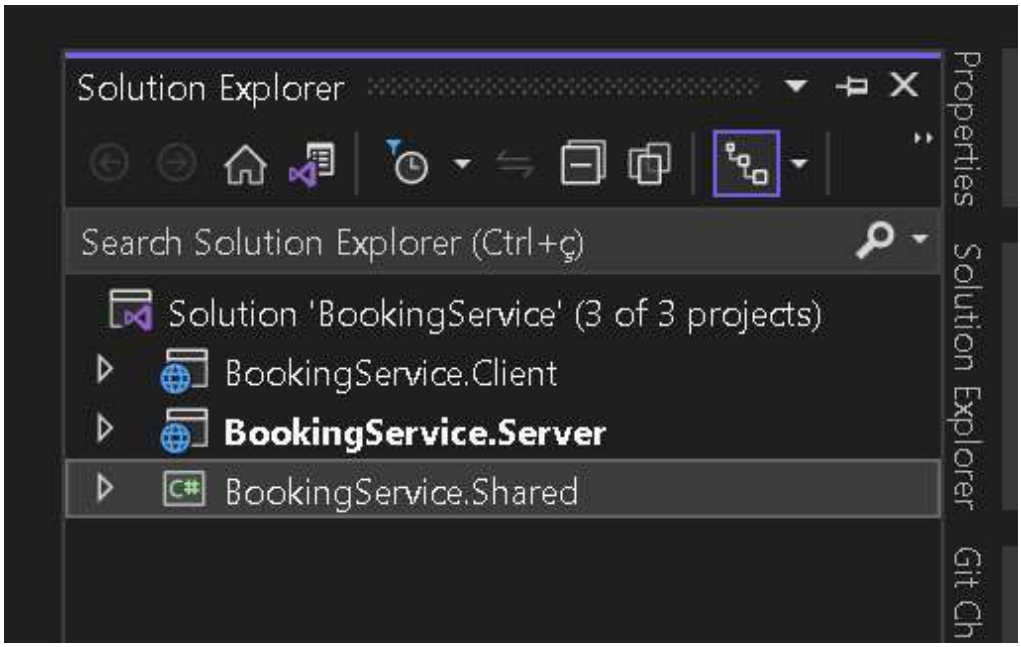


Figure 21.4: *Booking Service Solution*

These three projects are automatically created whenever the Blazor WebAssembly application is created with the Asp.Net Hosted option. The first project is the front-end one, the second project is the back-end, and the third project contains the code that is usually shared between all the projects in the Solution. In the context of this chapter, the Shared project will contain the model classes used by Entity Framework in the back end and by the forms and components in the front end.

Creating the models

As seen previously in this chapter, the scenario for the sample project requires the creation of models to support the basic operations that will be performed by a logistic company specialized in the transport of containers via maritime modal. The container type is the first model that needs to be created. Therefore, under the **`BookingService.Shared`** project in the Solution, create a class called **`ContainerType`**, with the properties shown in *Figure 21.5*:

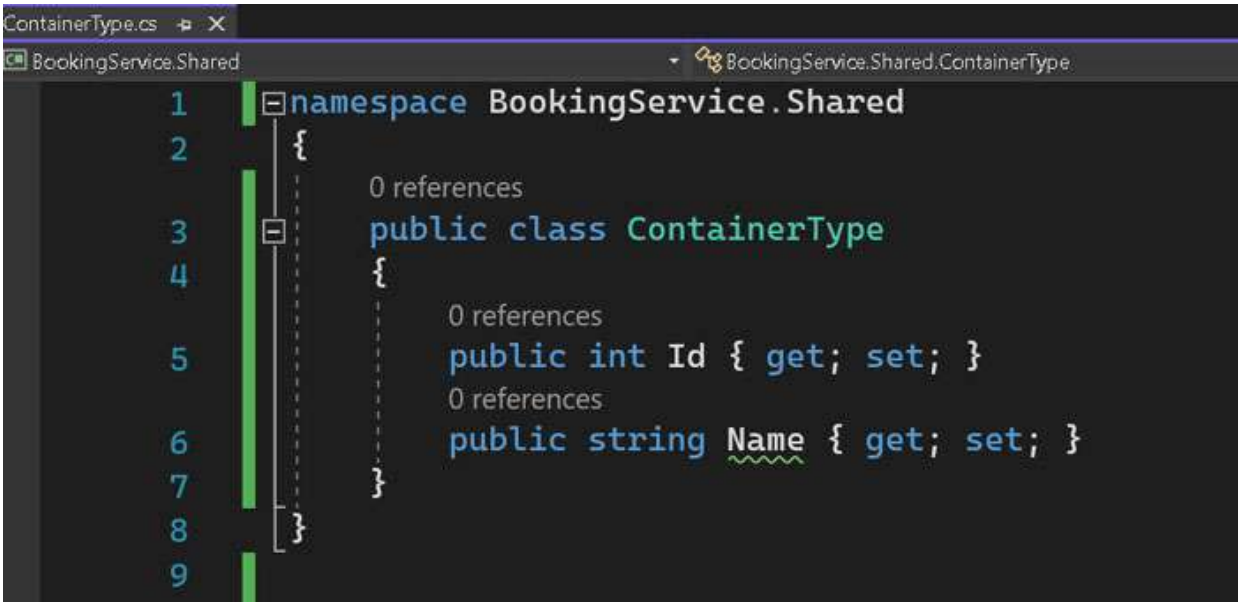


Figure 21.5: *Container Type class*

For simplification reasons, the **`ContainerType`** class has only two properties (ID and Name). All the namespaces are cleared, in this case from the class file, as none of them is being used apart from the global System namespace that is implicitly referenced for the entire project.

The next class is the Country one, which contains the properties ID, Name, and Initials, which need to be also created under the Shared project, as demonstrated in *Figure 21.6*:

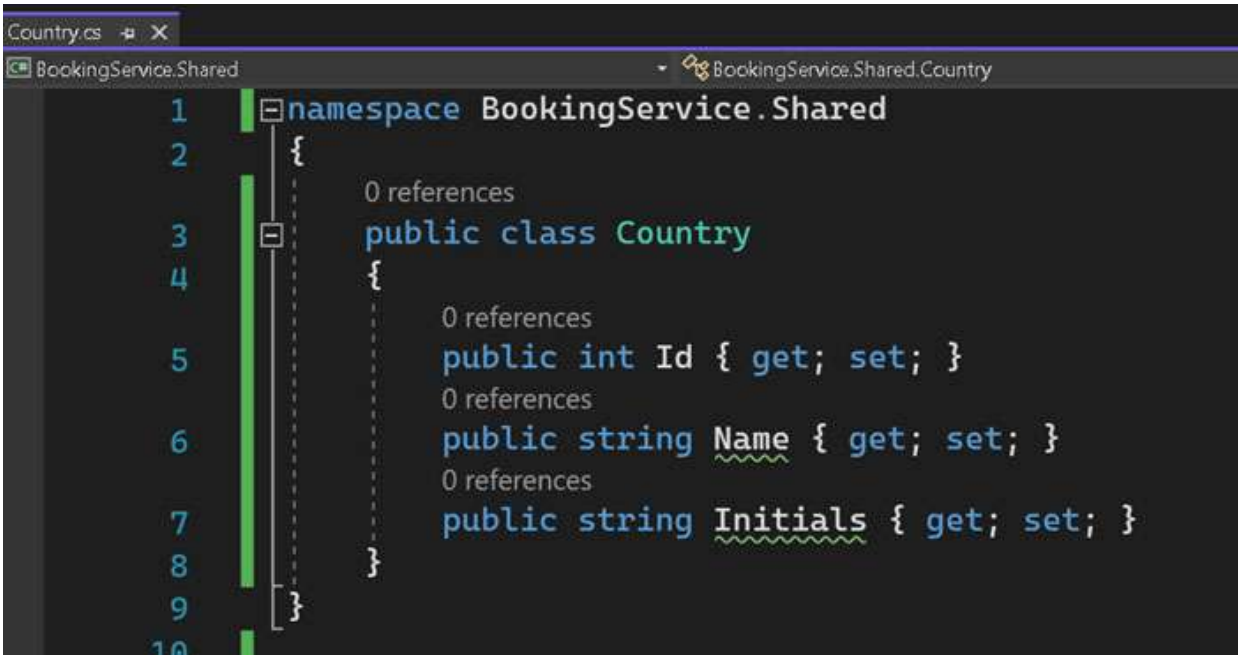


Figure 21.6: *Country class*

In this case, the Initials property will display the country's underlying flag on the front-end side using a CSS library. And this country class is used as a property in the Port model, which is represented in *Figure 21.7*:

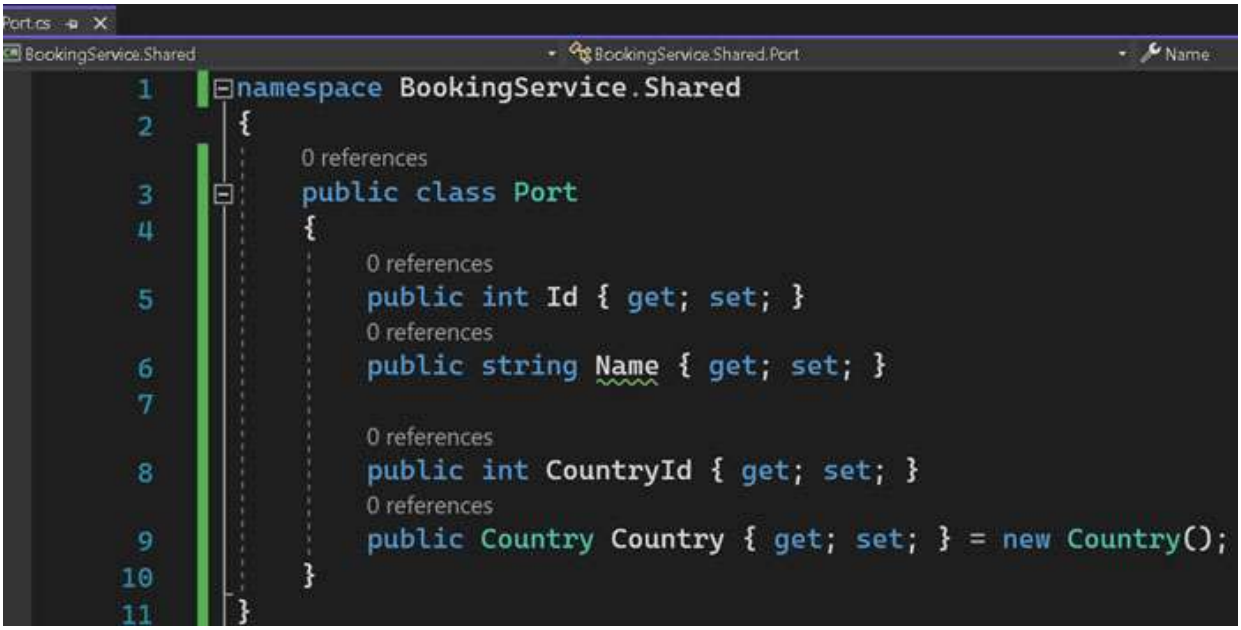


Figure 21.7: *Port class*

Remember that the Entity Framework will use all these classes in the project to generate the corresponding database tables. For that reason, the property **`CountryId`** is referenced in the Port model to represent the underlying foreign key later on.

Similar to the Port model, the Customer class also has a reference to the Country model, storing the fields **`Id`**, **`Name`**, **`CountryId`**, **`Country`**, Notes, and Active, as shown in *Figure 21.8*:

```
Customer.cs
BookingService.Shared                          BookingService.Shared.Customer
1   namespace BookingService.Shared
2   {
        0 references
3       public class Customer
4       {
            0 references
5           public int Id { get; set; }
            0 references
6           public string Name { get; set; }
            0 references
7           public string Address { get; set; }
            0 references
8           public int CountryId { get; set; }
            0 references
9           public Country Country { get; set; } = new Country();
            0 references
10          public string Notes { get; set; }
            0 references
11          public bool Active { get; set; }
12
13      }
14  }
```

Figure 21.8: *Customer class*

The Booking model represents the main model of the application, which has reference to all the other models and has the following properties: **`Id`**, **`CustomerId`**, **`Customer`**, **`ContainerTypeId`**, **`ContainerType`**, **`EmbarkDate`**, **`ArrivalDate`**, **`PortOriginId`**, **`PortOrigin`**, **`PortDestinyId`**, and **`PortDestiny`**, as shown in *Figure 21.9*. Note that the Port class is referenced twice in the Booking class, as the transport of the container involves two locations: origin and destination.

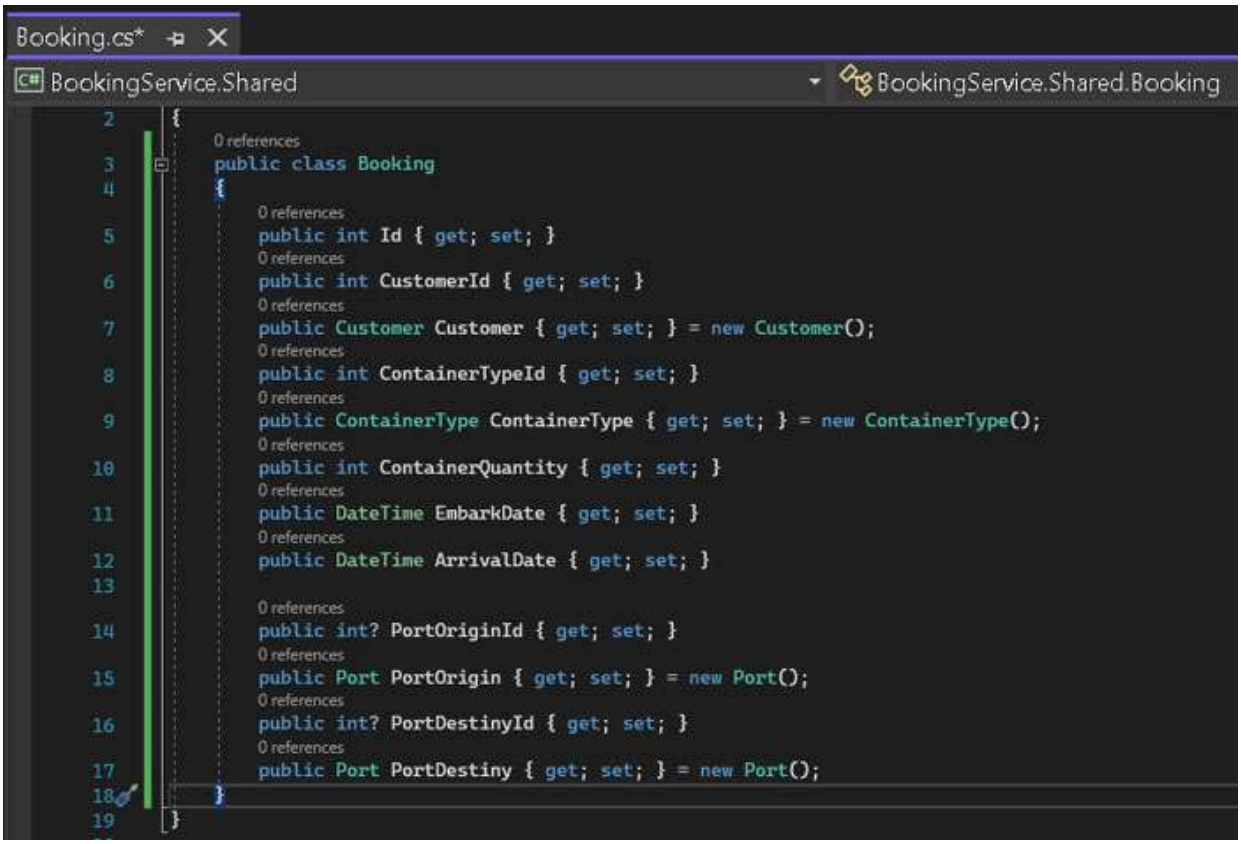


Figure 21.9: *Booking class*

Considering that all the models are already created in this stage, the next step of our example is creating the Database Context class for Entity Framework, which is explained in the next section.

Entity Framework configuration

As explained at the beginning of this chapter, this Booking Service project will use Entity Framework to generate all the database objects and establish all the basic CRUD operations in the system. The entire configuration for Entity Framework will be done in the **`BookingService.Server`** project. First of all, install all the packages related to Entity Framework Core, targeting the same .NET version for the project, as shown in *Figure 21.10*:

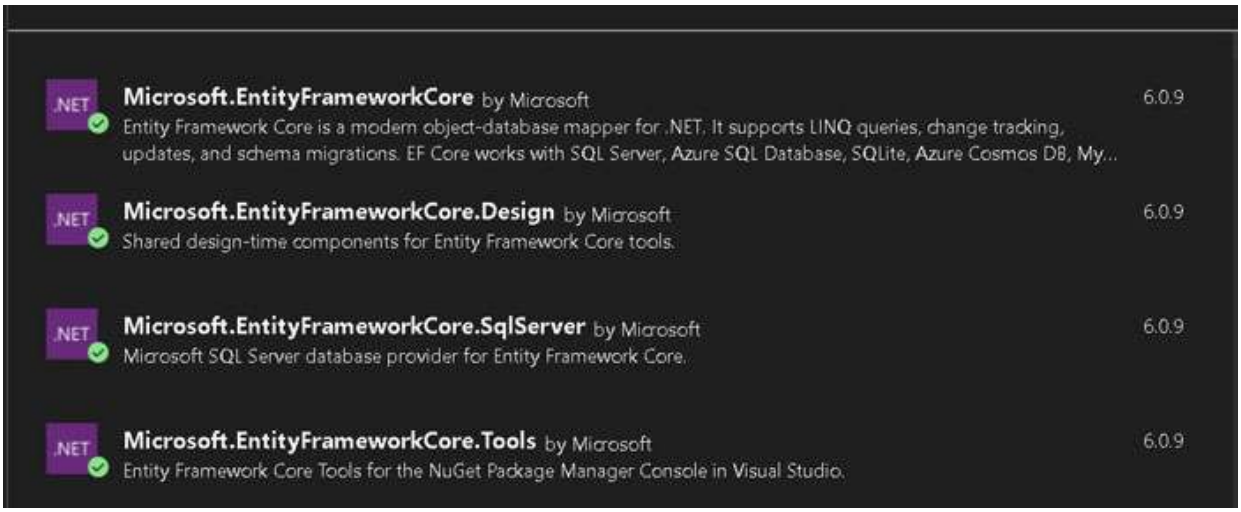


Figure 21.10: *Entity Framework packages*

All these packages allow us to use Entity Framework Core with the Code First approach, including the necessary middleware for SQL Server. This database is used during the development of the application of this chapter. You can host a SQL Server Express database in your local machine or a SQL Server instance in any cloud provider of your choice.

After installing all the Entity Framework packages, create a folder called "Data" in your **`BookingService.Server`** project and a class called **`SystemContext`** inside this folder. The **`SystemContext`** class will have the code and constructor as represented in *Figure 21.11*:

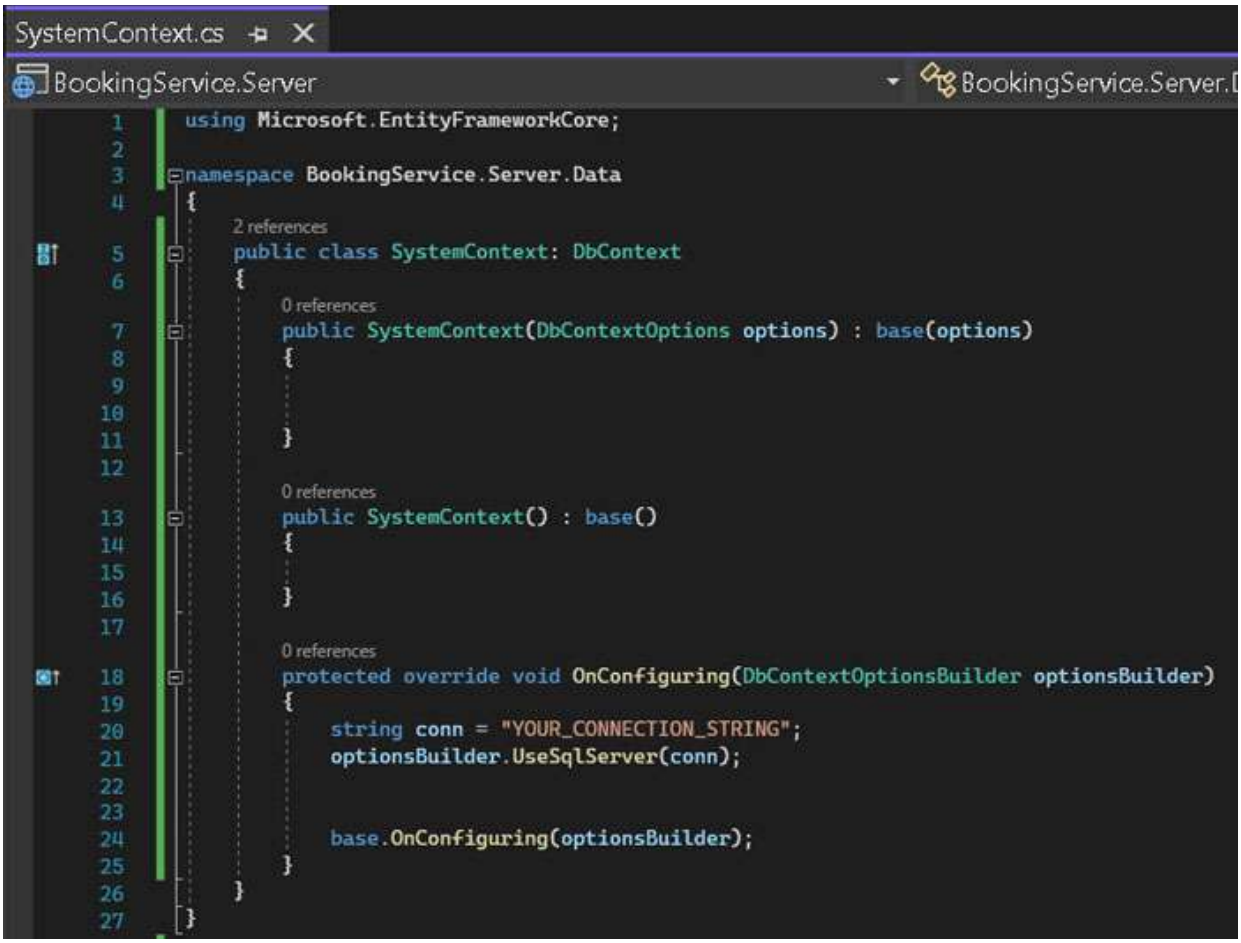


Figure 21.11: *SystemContext class*

Note that in the **`OnConfiguring`** method, the "conn" variable represents the connection string for your database, which could be a local SQL Server instance or a remote one. The **`SystemContext`** class inherits from the **`DbContext`** class, a native class from Entity Framework, and the next step is to create the actual DbSet properties for each one of the model classes for the Booking Service project (Booking, ContainerType, Country, Customer, and Port), as shown in *Figure 21.12*:

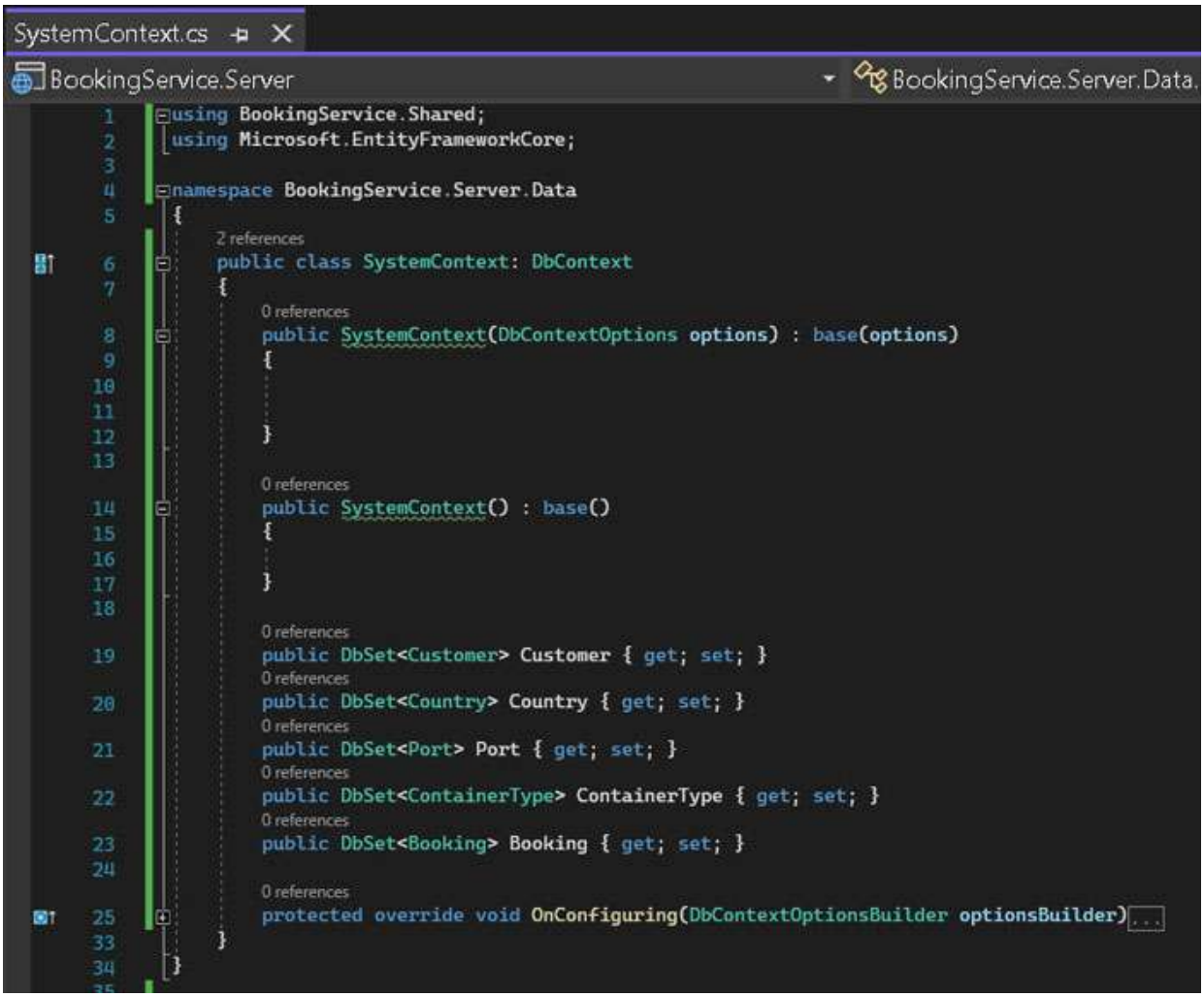


Figure 21.12: *SystemContext class*

When Entity Framework is being used in a .NET application, it is necessary to register the underlying service in the Startup of the application, which can be achieved by adding the highlighted code in the Program.cs file for the **`BookingService.Server`** project:

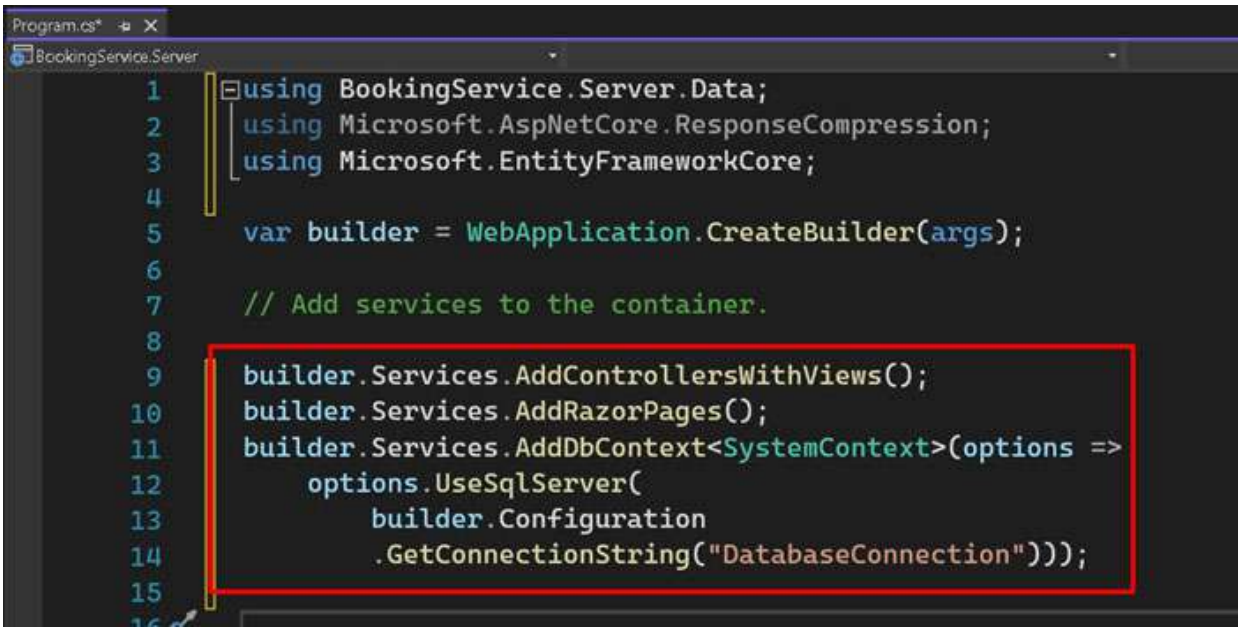


Figure 21.13: *Program.cs*

Note in line 14 that the Entity Framework settings get the connection string for the database from a configuration file. You can store the database in an application settings file, as an environment variable, or using any other method of your preference.

All the necessary configurations for the Entity Framework are already done in the **`BookingService`**. The server application, the next step is to generate the Entity Framework migration for each model specified in the **`SystemContext`** class as **`DbSet`**. This operation can be achieved by running the **`Add-Migration command`** in the Package Manager Console, as shown in *Figure 21.14*:

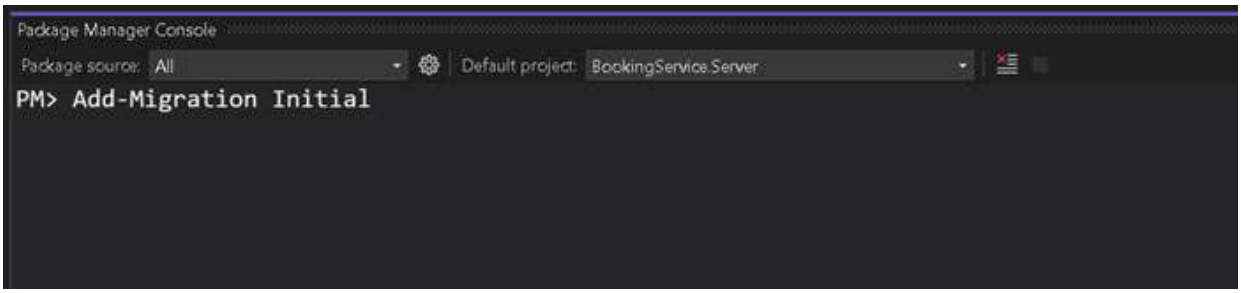


Figure 21.14: *Add-Migration command*

This command generates a folder called Migrations with a C# file that contains the code automatically created by Entity Framework that will handle the underlying database scripts, as highlighted in *Figure 21.15*:

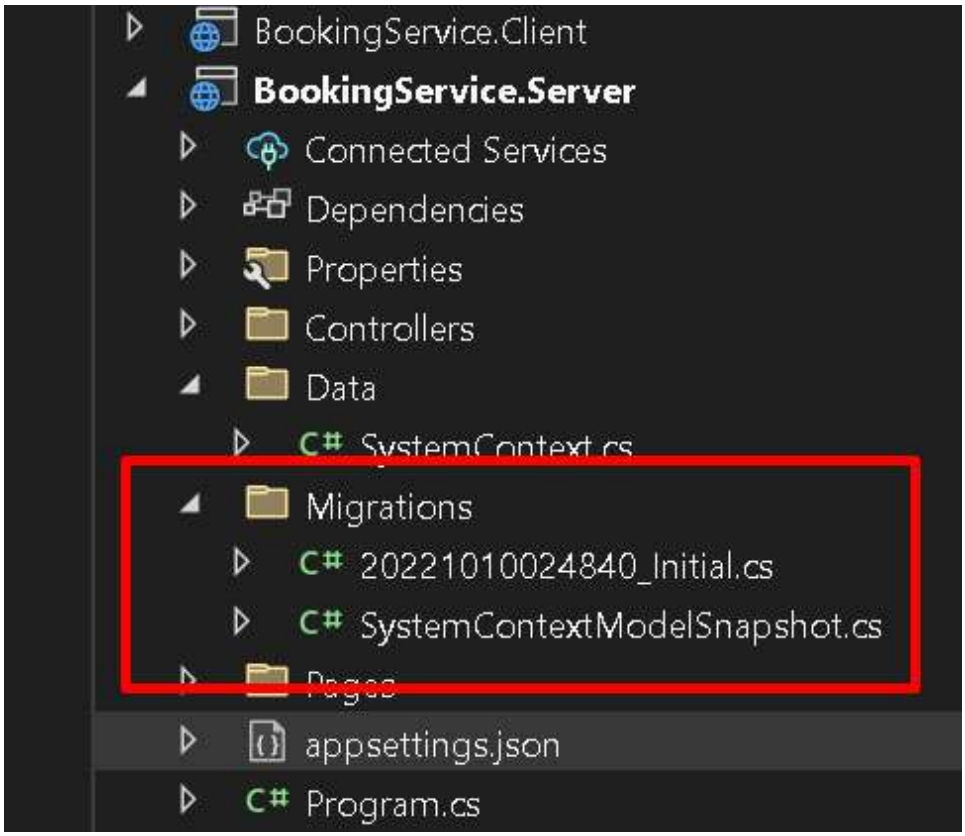


Figure 21.15: *Entity Framework migration*

If the connection string for the database is correctly set, all these steps will not throw any error. After generating the migration files, you can run the command "**`Update-Database`**," which executes the scripts in the underlying database, creating the actual database objects, as shown in *Figure 21.16*:

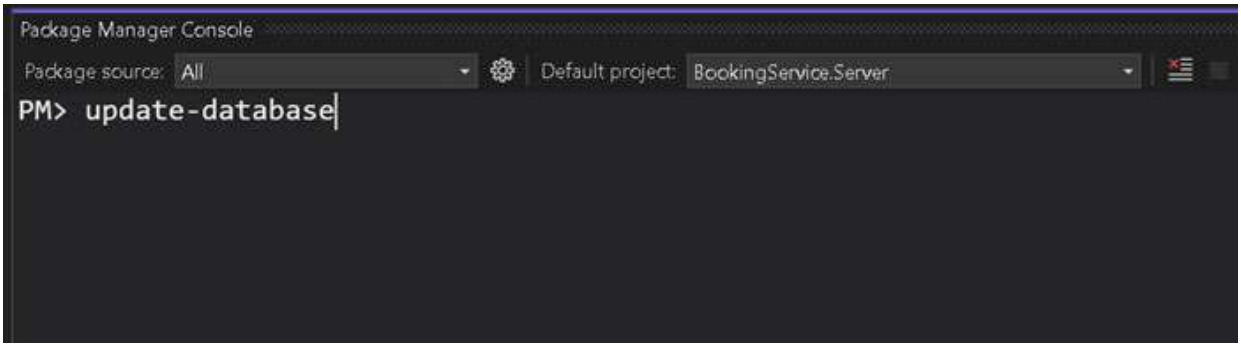


Figure 21.16: *Update-database command*

From this point, all the database objects exist in the SQL Server database. In the next section of this chapter, we will continue our journey in creating the

Booking Service application, creating the Business Logic class for each model of our application to separate the layers and concerns appropriately.

Creating the Business Logic layer

It is essential to have a correct separation between multiple layers in an enterprise application, which facilitates maintainability, testability, and reusability of classes, functions, and components. For studying purposes, the Business Logic layer of our Booking Service application will be placed inside a folder called “Logic.” Still, it could also be part of a separate project in the Solution. The first Business Logic class to be created is Container Type One. Each class inside the Logic folder will contain the following methods:

- **GetList**: This gets a list of the underlying entity.
- **Get**: This obtains information on a single entity.
- **Add**: This creates a new record in the database for the underlying entity type.
- **Edit**: This makes an update in the underlying entity.

Those are pretty basic operations performed by the system, representing everything we need to meet the requirements stated by the hypothetical scenario at the beginning of this chapter.

Create a folder called “Logic” in the `BookingService.Server` project and generate a class called `ContainerTypeLogic`, which includes the code shown in *Figure 21.17*:

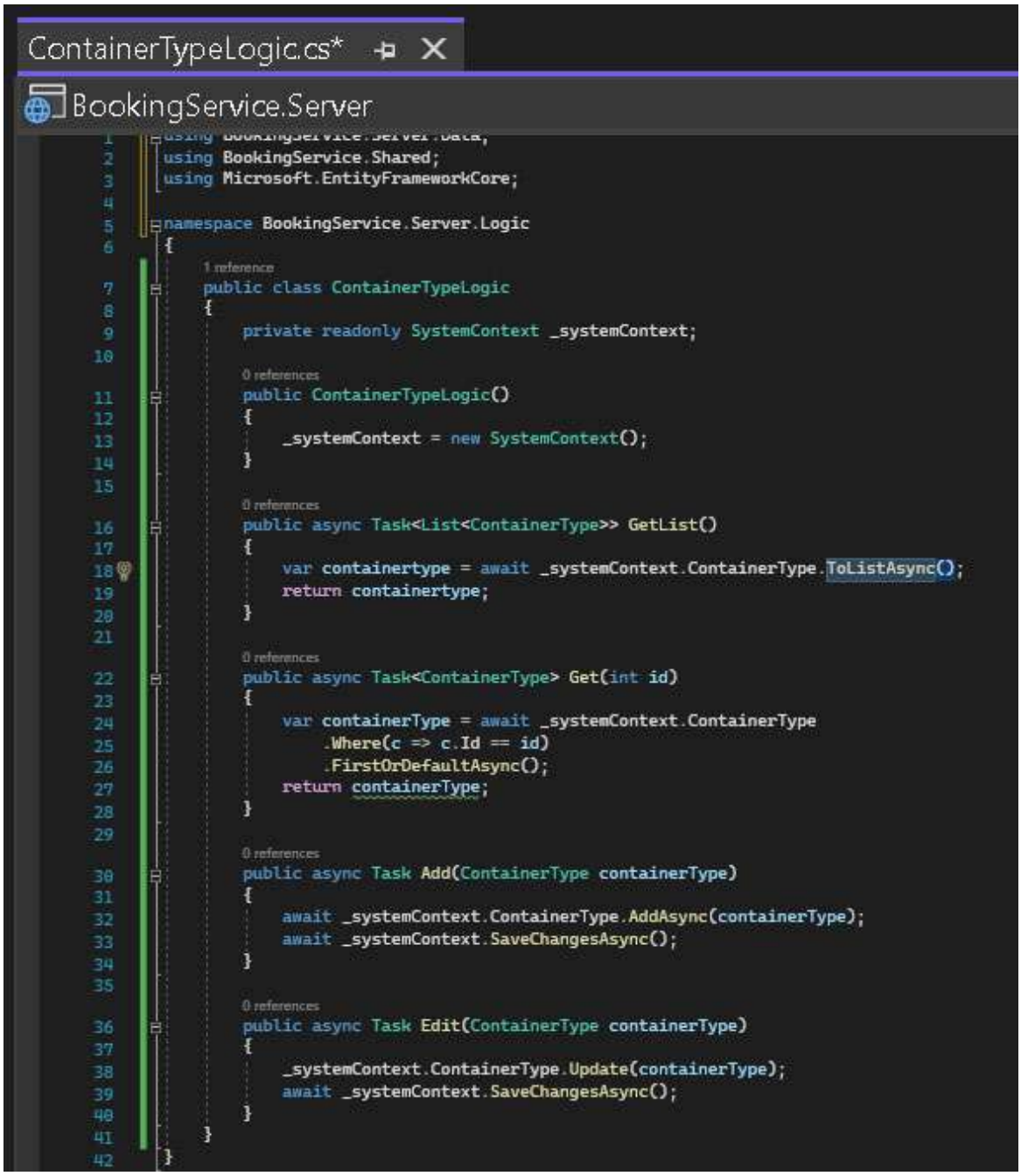


Figure 21.17: *ContainerTypeLogic class*

As seen in *Figure 21.17*, there are four methods in this class, one for each one stated in the requirements: **`GetList`**, **`Get`**, **`Add`**, and **`Edit`**. All the methods use an instance of the **`SystemContext`** class to perform operations in the database using **Entity Framework**. Additionally, the Add and Edit methods use the **`SaveChangesAsync`** method to confirm the database transaction for the underlying operations.

Similarly, to the ContainerTypeLogic class, create a **CountryLogic** class with the code represented in *Figure 21.18*:

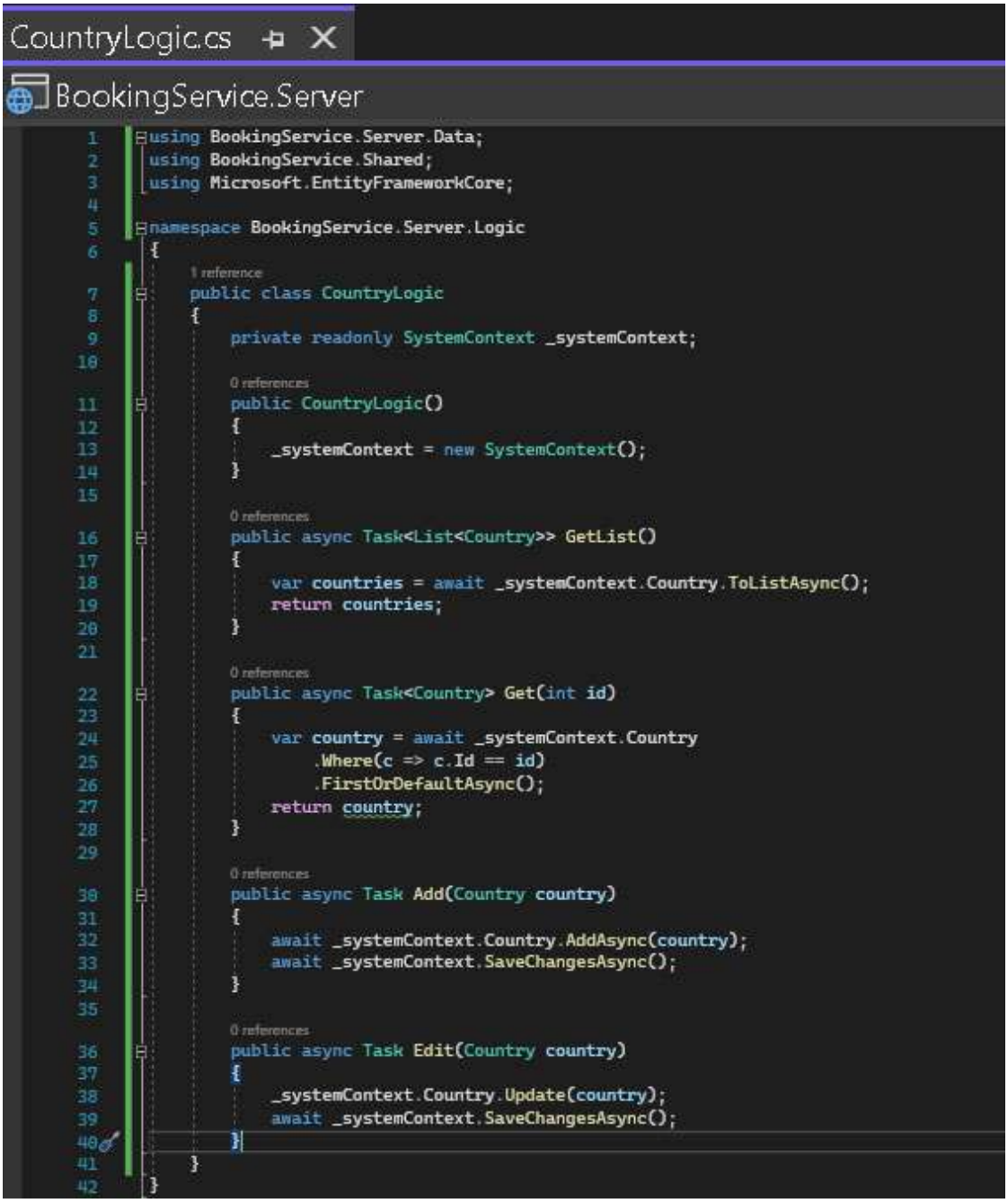


Figure 21.18: *CountryLogic class*

Repeat the same operation for the **`CustomerLogic`** class, which has a handler in the Add and Edit methods to set the Country property as null, as the Entity Framework uses the **`CountryId`** property to set the Country information associated with the Customer entity, as shown in *Figure 21.19:*

CustomerLogic.cs

BookingService.Server

```
using BookingService.Server.Data;
using BookingService.Shared;
using Microsoft.EntityFrameworkCore;

namespace BookingService.Server.Logic
{
    1 reference
    public class CustomerLogic
    {
        private readonly SystemContext _systemContext;

        0 references
        public CustomerLogic()
        {
            _systemContext = new SystemContext();
        }

        0 references
        public async Task<List<Customer>> GetList()
        {
            var customers = await _systemContext.Customer
                .Include(x => x.Country)
                .ToListAsync();

            return customers;
        }

        0 references
        public async Task<Customer> Get(int id)
        {
            var customer = await _systemContext.Customer
                .Where(c => c.Id == id)
                .FirstOrDefaultAsync();

            return customer;
        }

        0 references
        public async Task Add(Customer customer)
        {
            customer.Country = null;
            await _systemContext.Customer.AddAsync(customer);
            await _systemContext.SaveChangesAsync();
        }

        0 references
        public async Task Edit(Customer customer)
        {
            customer.Country = null;
            _systemContext.Customer.Update(customer);
            await _systemContext.SaveChangesAsync();
        }
    }
}
```

Figure 21.19: *CustomerLogic class*

The underlying code for the **`PortLogic`** class, our next file to be created, follows a similar structure, setting the Country property as null in the Add

and Edit methods, as shown in *Figure 21.20*:

PortLogic.cs

BookingService.Server

```
using BookingService.Server.Data;
using BookingService.Shared;
using Microsoft.EntityFrameworkCore;

namespace BookingService.Server.Logic
{
    1 reference
    public class PortLogic
    {
        private readonly SystemContext _systemContext;

        0 references
        public PortLogic()
        {
            _systemContext = new SystemContext();
        }

        0 references
        public async Task<List<Port>> GetList()
        {
            var ports = await _systemContext.Port
                .Include(x => x.Country)
                .ToListAsync();

            return ports;
        }

        0 references
        public async Task<Port> Get(int id)
        {
            var port = await _systemContext.Port
                .Where(c => c.Id == id)
                .FirstOrDefaultAsync();
            return port;
        }

        0 references
        public async Task Add(Port port)
        {
            port.Country = null;
            await _systemContext.Port.AddAsync(port);
            await _systemContext.SaveChangesAsync();
        }

        0 references
        public async Task Edit(Port port)
        {
            port.Country = null;
            _systemContext.Port.Update(port);
            await _systemContext.SaveChangesAsync();
        }
    }
}
```

Figure 21.20: *PortLogic class*

Finally, the last Business Logic class is the **`BookingLogic`** one, which has the code shown in *Figure 21.21*:

```
namespace BookingService.Server.Logic
{
    1 reference
    public class BookingLogic
    {
        private readonly SystemContext _systemContext;

        0 references
        public BookingLogic()
        {
            _systemContext = new SystemContext();
        }

        0 references
        public async Task<List<Booking>> GetList()
        {
            var bookings = await _systemContext.Booking
                .Include(x => x.Customer)
                .Include(x => x.PortOrigin)
                .Include(x => x.PortOrigin.Country)
                .Include(x => x.PortDestiny)
                .Include(x => x.PortDestiny.Country)
                .ToListAsync();

            return bookings;
        }

        0 references
        public async Task<Booking> Get(int id)
        {
            var booking = await _systemContext.Booking
                .Where(c => c.Id == id)
                .FirstOrDefaultAsync();

            return booking;
        }

        0 references
        public async Task Add(Booking booking)
        {
            booking.Customer = null;
            booking.PortOrigin = null;
            booking.PortDestiny = null;
            booking.ContainerType = null;

            await _systemContext.Booking.AddAsync(booking);
            await _systemContext.SaveChangesAsync();
        }

        0 references
        public async Task Edit(Booking booking)
        {
            booking.Customer = null;
            booking.PortOrigin = null;
            booking.PortDestiny = null;
            booking.ContainerType = null;

            _systemContext.Booking.Update(booking);
            await _systemContext.SaveChangesAsync();
        }
    }
}
```

Figure 21.21: *BookingLogic class*

Note in the **`BookingLogic`** class that the properties **`ContainerType`**, **`PortOrigin`**, **`PortDestinty`**, and Customer are set to null in the Add and Edit methods for the same reason that is done in the other classes.

Creating the Controllers

Considering the front-end application will be consuming resources from the back-end via Web API, it is necessary to create a Controller for each one of the models in the application. Inside the Controllers folder in the **`BookingService.Server`** project, create a **`ContainerTypeController`** class with the code shown in *Figure 21.22*:

ContainerTypeController.cs

BookingService.Server

```
using BookingService.Server.Logic;
using BookingService.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BookingService.Server.Controllers
{
    [ApiController]
    [Route("[controller]/[action]")]
    3 references
    public class ContainerTypeController : ControllerBase
    {
        private readonly ILogger<ContainerTypeController> _logger;
        private readonly ContainerTypeLogic _containerTypeLogic;
        0 references
        public ContainerTypeController(ILogger<ContainerTypeController> logger)
        {
            _logger = logger;
            _containerTypeLogic = new ContainerTypeLogic();
        }

        [HttpGet]
        0 references
        public async Task<List<ContainerType>> GetList()
        {
            return await _containerTypeLogic.GetList();
        }

        [HttpGet]
        0 references
        public async Task<ContainerType> Get(int id)
        {
            return await _containerTypeLogic.Get(id);
        }

        [HttpPost]
        0 references
        public async Task Add([FromBody] ContainerType containerType)
        {
            await _containerTypeLogic.Add(containerType);
        }

        [HttpPost]
        0 references
        public async Task Edit([FromBody] ContainerType containerType)
        {
            await _containerTypeLogic.Edit(containerType);
        }
    }
}
```

Figure 21.22: *ContainerTypeController class*

The **`ContainerTypeController`** class and any other Controller classes in the project follow the same structure: a corresponding Controller method for each operation existing in the underlying Business Logic class. The **`GetList`** and Get methods use GET as HTTP Verb, and the Add and Edit methods use the POST HTTP Verb.

The Controller for the Country model has an equal representation, as shown in *Figure 21.23*:

CountryController.cs

BookingService.Server

```
using BookingService.Server.Logic;
using BookingService.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BookingService.Server.Controllers
{
    [ApiController]
    [Route("[controller]/[action]")]
    3 references
    public class CountryController : ControllerBase
    {
        private readonly ILogger<CountryController> _logger;
        private readonly CountryLogic _countryLogic;
        0 references
        public CountryController(ILogger<CountryController> logger)
        {
            _logger = logger;
            _countryLogic = new CountryLogic();
        }

        [HttpGet]
        0 references
        public async Task<List<Country>> GetList()
        {
            return await _countryLogic.GetList();
        }

        [HttpGet]
        0 references
        public async Task<Country> Get(int id)
        {
            return await _countryLogic.Get(id);
        }

        [HttpPost]
        0 references
        public async Task Add([FromBody] Country country)
        {
            await _countryLogic.Add(country);
        }

        [HttpPost]
        0 references
        public async Task Edit([FromBody] Country country)
        {
            await _countryLogic.Edit(country);
        }
    }
}
```

Figure 21.23: *CountryController class*

A private property for the Business Logic class reference is specified at the Controller's beginning and populated in the constructor. If needed, dependency injection in the back-end project could be used to facilitate integration tests.

The next Controller is the **`PortInfoController`**, which contains the endpoints that call the underlying Port Logic class directly, as shown in *Figure 21.24*:

PortInfoController.cs

BookingService.Server

```
using BookingService.Server.Logic;
using BookingService.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BookingService.Server.Controllers
{
    [ApiController]
    [Route("[controller]/[action]")]
    3 references
    public class PortInfoController : ControllerBase
    {
        private readonly ILogger<PortInfoController> _logger;
        private readonly PortLogic _portLogic;
        0 references
        public PortInfoController(ILogger<PortInfoController> logger)
        {
            _logger = logger;
            _portLogic = new PortLogic();
        }

        [HttpGet]
        0 references
        public async Task<List<Port>> GetList()
        {
            return await _portLogic.GetList();
        }

        [HttpGet]
        0 references
        public async Task<Port> Get(int id)
        {
            return await _portLogic.Get(id);
        }

        [HttpPost]
        0 references
        public async Task Add([FromBody] Port port)
        {
            await _portLogic.Add(port);
        }

        [HttpPost]
        0 references
        public async Task Edit([FromBody] Port port)
        {
            await _portLogic.Edit(port);
        }
    }
}
```

Figure 21.24: *PortInfoController class*

Note that the number of Controllers is identical to the models in the application, as the front end will contain screens that refer to each of these

models, retrieving information from the back end. Keeping the Controllers separated by type is helpful to improve testability and other aspects regarding the **Single Responsibility Principle** (**SRP**) stated by the SOLID principles. Given that the next Controller is related to the Customer model, which has the code represented in *Figure 21.25*:

CustomerInfoController.cs

BookingService.Server

```
using BookingService.Server.Logic;
using BookingService.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BookingService.Server.Controllers
{
    [ApiController]
    [Route("[controller]/[action]")]
    3 references
    public class CustomerInfoController : ControllerBase
    {
        private readonly ILogger<CustomerInfoController> _logger;
        private readonly CustomerLogic _customerLogic;
        0 references
        public CustomerInfoController(ILogger<CustomerInfoController> logger)
        {
            _logger = logger;
            _customerLogic = new CustomerLogic();
        }

        [HttpGet]
        0 references
        public async Task<List<Customer>> GetList()
        {
            return await _customerLogic.GetList();
        }

        [HttpGet]
        0 references
        public async Task<Customer> Get(int id)
        {
            return await _customerLogic.Get(id);
        }

        [HttpPost]
        0 references
        public async Task Add([FromBody] Customer customer)
        {
            await _customerLogic.Add(customer);
        }

        [HttpPost]
        0 references
        public async Task Edit([FromBody] Customer customer)
        {
            await _customerLogic.Edit(customer);
        }
    }
}
```

Figure 21.25: *CustomerInfoController class*

Finally, the last Controller that needs to be created is the one related to Booking information, which has the code represented in *Figure 21.26*:

BookingInfoController.cs

BookingService.Server

```
using BookingService.Server.Logic;
using BookingService.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BookingService.Server.Controllers
{
    [ApiController]
    [Route("[controller]/[action]")]
    3 references
    public class BookingInfoController : ControllerBase
    {
        private readonly ILogger<BookingInfoController> _logger;
        private readonly BookingLogic _bookingLogic;
        0 references
        public BookingInfoController(ILogger<BookingInfoController> logger)
        {
            _logger = logger;
            _bookingLogic = new BookingLogic();
        }

        [HttpGet]
        0 references
        public async Task<List<Booking>> GetList()
        {
            return await _bookingLogic.GetList();
        }

        [HttpGet]
        0 references
        public async Task<Booking> Get(int id)
        {
            return await _bookingLogic.Get(id);
        }

        [HttpPost]
        0 references
        public async Task Add([FromBody] Booking booking)
        {
            await _bookingLogic.Add(booking);
        }

        [HttpPost]
        0 references
        public async Task Edit([FromBody] Booking booking)
        {
            await _bookingLogic.Edit(booking);
        }
    }

}
```

Figure 21.26: *BookingInfoController class*

At this stage, all the necessary Controllers for the Web API are already created and ready to be consumed by the front-end application based on

Blazor WebAssembly. In the next section, you will have the opportunity to experience the creation of all the necessary components in the front-end application, including using Blazorise, an open-source UI framework for Blazor applications.

Creating the front-end

The front-end application will use Razor Components to render Web pages based on WebAssembly, taking all the benefits of Blazor and reusing the models from the **`BookingService`**. The shared project that was used to create the Entity Framework DbSets. As shown in *Figure 21.27*, the default template for Blazor WebAssembly applications already contains specific pages:

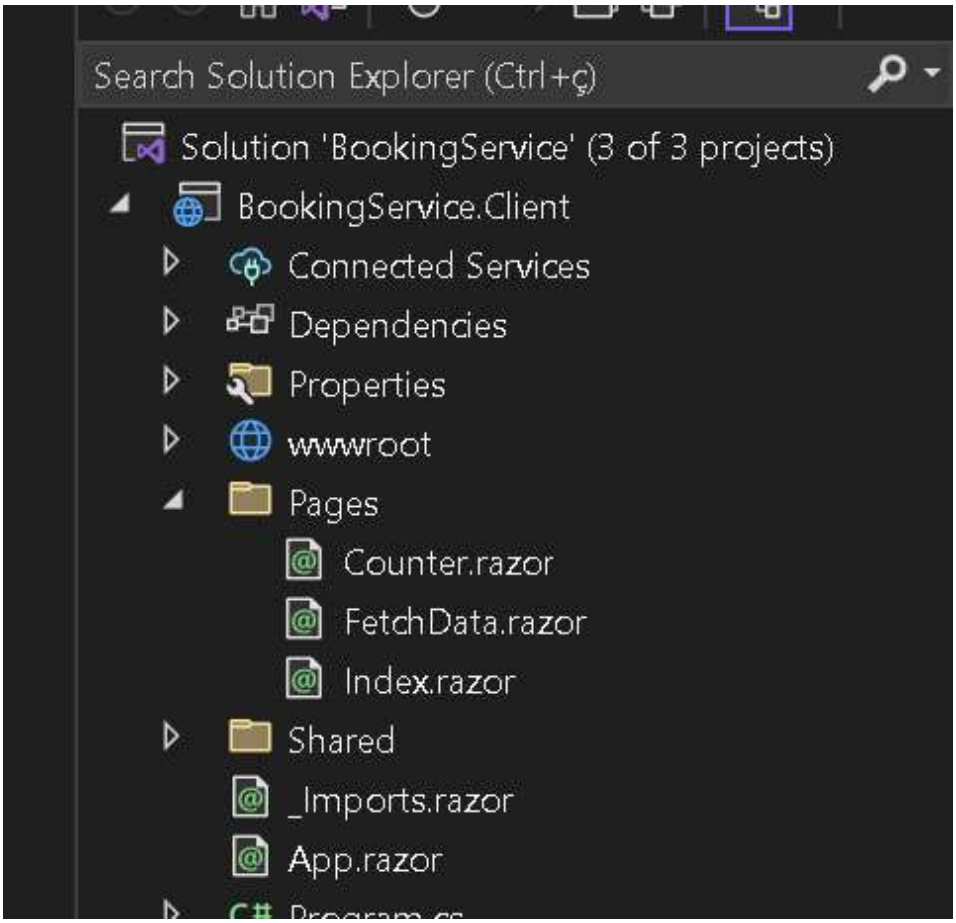


Figure 21.27: *BookingInfoController class*

In the context of the project demonstrated in this chapter, only the **`Index.razor`** component is kept. But before creating other components,

ensure you have all the necessary packages installed in the **`BookingService.Client`** project, which is listed in *Figure 21.28*:

```
<PackageReference Include="Blazorise.Bootstrap" Version="1.1.1" />
<PackageReference Include="Blazorise.Icons.FontAwesome" Version="1.1.1" />
<PackageReference Include="Microsoft.AspNet.WebApi.Client" Version="5.2.9" />
<PackageReference Include="Microsoft.AspNetCore.Components" Version="6.0.9" />
<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="6.0.7" />
<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer" Version="6.0.7" PrivateAssets="all" />
<PackageReference Include="Newtonsoft.Json.Bson" Version="1.0.2" />
```

Figure 21.28: *Nuget packages*

After installing all these packages, include the necessary references to the Blazorise UI framework in the Program.cs file, installed via Nuget Package, as highlighted in *Figure 21.29:*

```
Program.cs
BookingService.Client
using Blazorise.Bootstrap;
using Blazorise.Icons.FontAwesome;
using BookingService.Client;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

builder.Services.AddScoped(sp => new HttpClient { BaseAddress =
    new Uri(builder.HostEnvironment.BaseAddress) });

builder.Services
    .AddBlazorise(options =>
    {
        options.Immediate = true;
    })
    .AddBootstrapProviders()
    .AddFontAwesomeIcons();

await builder.Build().RunAsync();
```

Figure 21.29: *Program.cs*

As the front-end application will use a CSS library to render flags for the countries, make the following change in the **`index.html`** file presented within the wwwroot folder, as highlighted in *Figure 21.30:*

Figure 21.30: *Index.html*

Each one of the models will have two pages in the front-end applications: one of them to list all the records from the back-end and another to add and update records. The next section will show you how to create the underlying pages for the Container Type model.

Container type components

First of all, as we are using Blozorise as a UI framework, it is necessary to specify the underlying reference in the **`_Imports.razor`** file to make the corresponding components from the framework available across all components of the Blazor WebAssembly application, as shown in *Figure 21.31*:

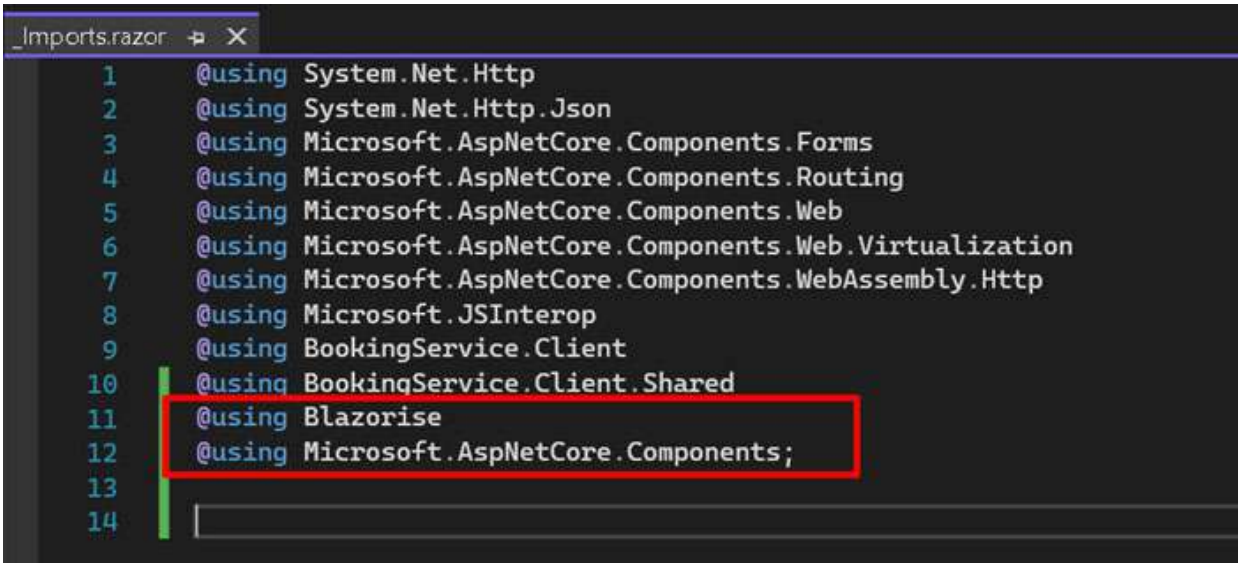


Figure 21.31: *_Imports.razor*

This file contains the global references for the project. The next step is to create the **`ContainerTypes.razor`** component inside the Pages folder. Each Razor component has two main sections: one that is exclusive for HTML,

CSS, JS, and Razor code and another section called “code,” which contains references to the C# code that includes logical statements associated with the component. After creating the razor corresponding razor component, include the method **`OnInitializedAsync`** method and a private **`containerTypeList`** property in the code section, as shown in *Figure 21.32*:

```
@code {
    private List<ContainerType>? containerTypesList;

    protected override async Task OnInitializedAsync()
    {
        var result = await Http.GetAsync("/Containertype/GetList");
        var stringResult = await result.Content.ReadAsStringAsync();
        containerTypesList = JsonConvert.DeserializeObject<List<ContainerType>>(stringResult);
    }
}
```

Figure 21.32: *Container type code section*

The **`OnInitializedAsync`** method makes an HTTP request to the endpoint we previously created in the back-end in the **`ContainerTypeController`** file. After making the request, the response is parsed to a list of container types.

The next step is to create the actual HTML that will list the container types populated in the code section. The necessary adaptation in the component should look like the code in *Figure 21.33*:

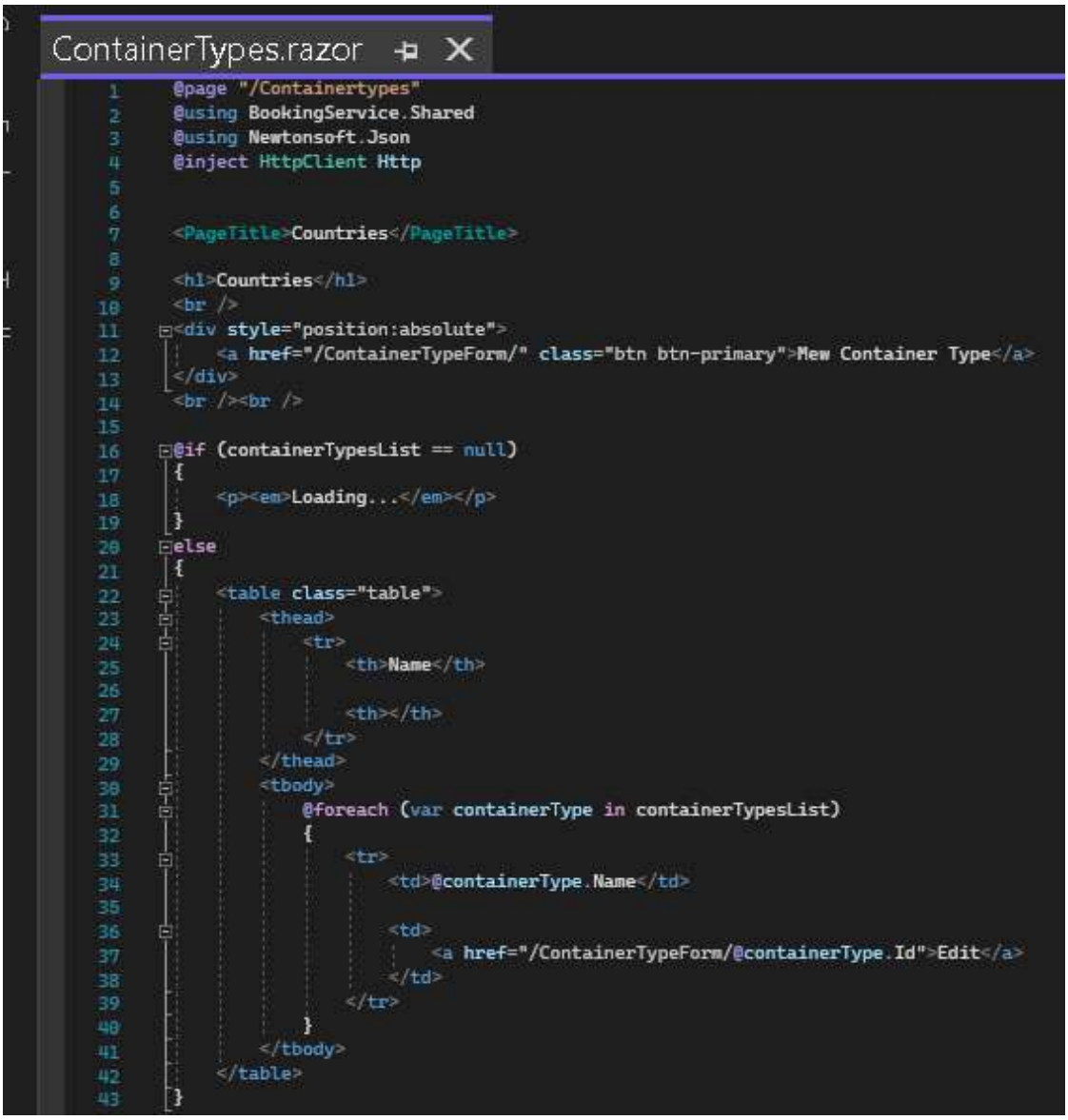


Figure 21.33: *Container type HTML section*

In this component, the route "**`/ContainerTypes`**" is specified in the first line using the @page directive, which means this is the address to access the container type list page. In line 4, an HttpClient object is being injected, which is a good practice to reuse the same instance multiple times. If you are familiar with Asp.Net MVC with Razor Pages, you note that the syntax is identical in terms of creating loops and rendering the dynamic information. In the last column of the HTML table, there is a link to redirect to the **`ContainerTypeForm`**, which will be created in the following.

Create a new Razor component called **`ContainerTypeForm.razor`**, and include the following content in the code section, as demonstrated in *Figure*

21.34:

```
31  @code {
32      private ContainerType containerType;
33
34      [Parameter]
35      public string ContainerTypeId { get; set; }
36
37      protected override async Task OnInitializedAsync()
38      {
39          if(ContainerTypeId != null)
40          {
41              var result = await Http.GetAsync("/ContainerType/Get?id=" + ContainerTypeId);
42              var stringResult = await result.Content.ReadAsStringAsync();
43              containerType = JsonConvert.DeserializeObject<ContainerType>(stringResult);
44          }
45          else
46          {
47              containerType = new ContainerType();
48          }
49
50      }
51
52      protected async Task SaveAsync()
53      {
54          var myContent = JsonConvert.SerializeObject(containerType);
55          var buffer = System.Text.Encoding.UTF8.GetBytes(myContent);
56          var stringContent = new StringContent(myContent);
57
58
59          if(containerType.Id == 0)
60          {
61              await Http.PostAsJsonAsync("/ContainerType/Add", containerType);
62          }
63          else
64          {
65              await Http.PostAsJsonAsync("/ContainerType/Edit", containerType);
66          }
67
68          NavManager.NavigateTo("/containerTypes");
69
70      }
71  }
72
```

Figure 21.34: *Container Type Form code section*

In line 32, the private **`ContainerType`** property is specified, representing the model the form will handle. Within this code, there is a logical statement to get the corresponding Container Type from the back end if the ID is passed as a parameter. Additionally, the SaveAsync method contains a condition to call different endpoints based on the Add and Edit operations.

As we already have the logical implementation at this stage, we can create the visual part of our component, which is demonstrated in *Figure 21.35*:

ContainerTypeForm.razor

```
@page "/ContainerTypeForm/{ContainertypeId?}"
@using BookingService.Shared
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject HttpClient Http
@inject NavigationManager NavManager


<PageTitle>Container Type Form</PageTitle>

<h1>Container Type Form</h1>

@if(containerType != null)
{

        <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Name" @bind-Text="@containerType.Name" />
            <Feedback>
                <ValidationError>Enter valid name!</ValidationError>
            </Feedback>
            </Validation>
        </Field>

         <Button Color="Color.Primary" Clicked="@SaveAsync">Save</Button>


}
```

Figure 21.35: *Container Type visual section*

The Razor section of the component uses the Field, Validation, Button, and TextEdit components from the **`Blazorise`** framework. In this case, the form warns the user if the Name field's conditions are not satisfied. The Save button calls the **`SaveAsync`** method created previously. The next step is to specify a link to the new pages in the **`NavMenu`** component, which is inside the Shared folder, as shown in *Figure 21.36*:

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="containerTypes">
                <span class="oi oi-plus" aria-hidden="true"></span> Container Types
            </NavLink>
        </div>

    </nav>
</div>
```

Figure 21.36: *Menu option for Container Types*

If you run the application, you will be able to navigate to the container types page via the lateral menu on the left, as shown in *Figure 21.37*:

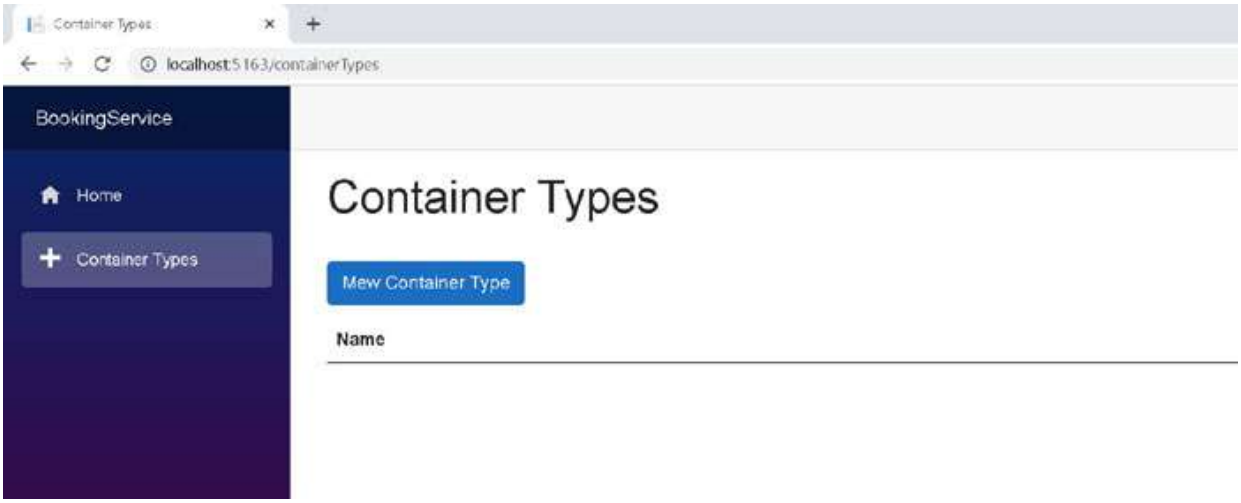


Figure 21.37: *Container types page*

It is essential to highlight that, in some cases, your browser may cache the JavaScript files generated by Visual Studio in runtime, generating an error with the message “Failed to find a valid digest in the integrity attribute for the resource on IIS.” If this error appears, you will have to clean your browser's cache.

If you click on the **`New Container Type`** button, you will be redirected to the Container Type Form and will be able to create a new entry in the database, as shown in *Figure 21.38*:

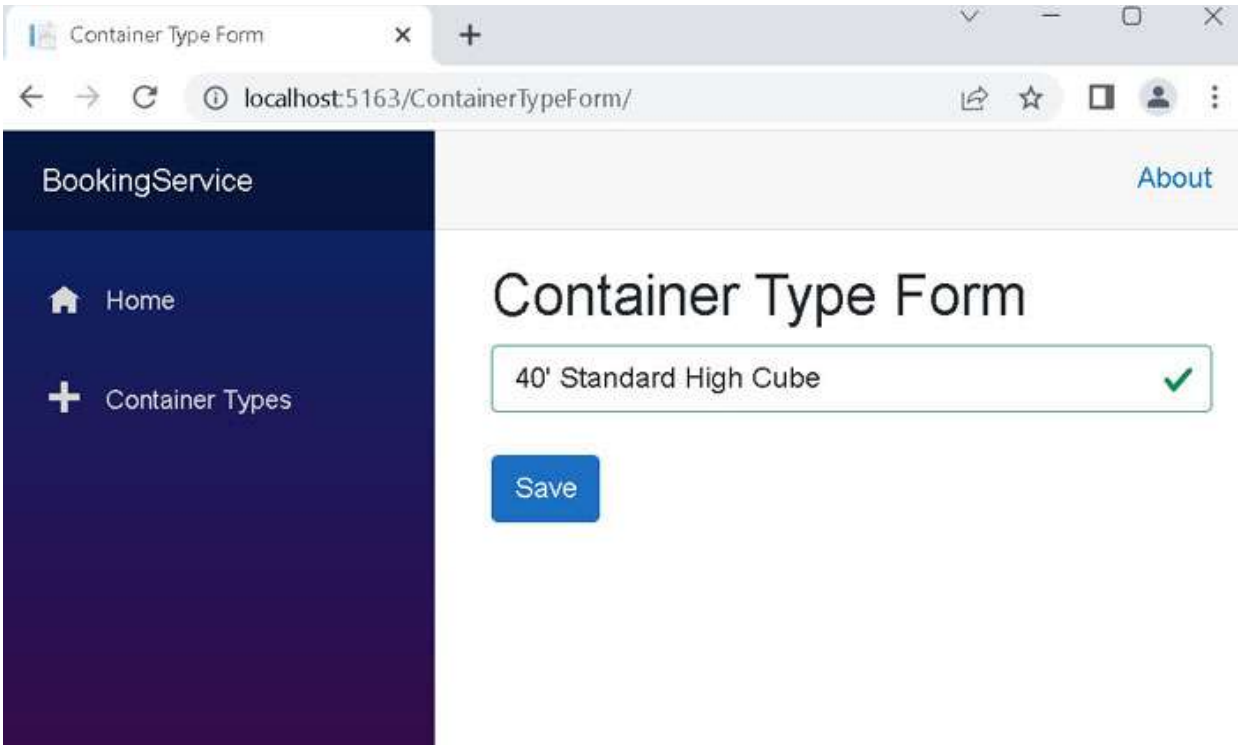


Figure 21.38: *Container type form page*

If the validation passes, the field is marked with green color. After saving the new container type, the application redirects to the container types list page again, which is the behavior specified in the **`SaveAsync`** method. With a single record in the database, you will be able to see the entry in the list and will be able to edit the container type, as shown in *Figure 21.39*:

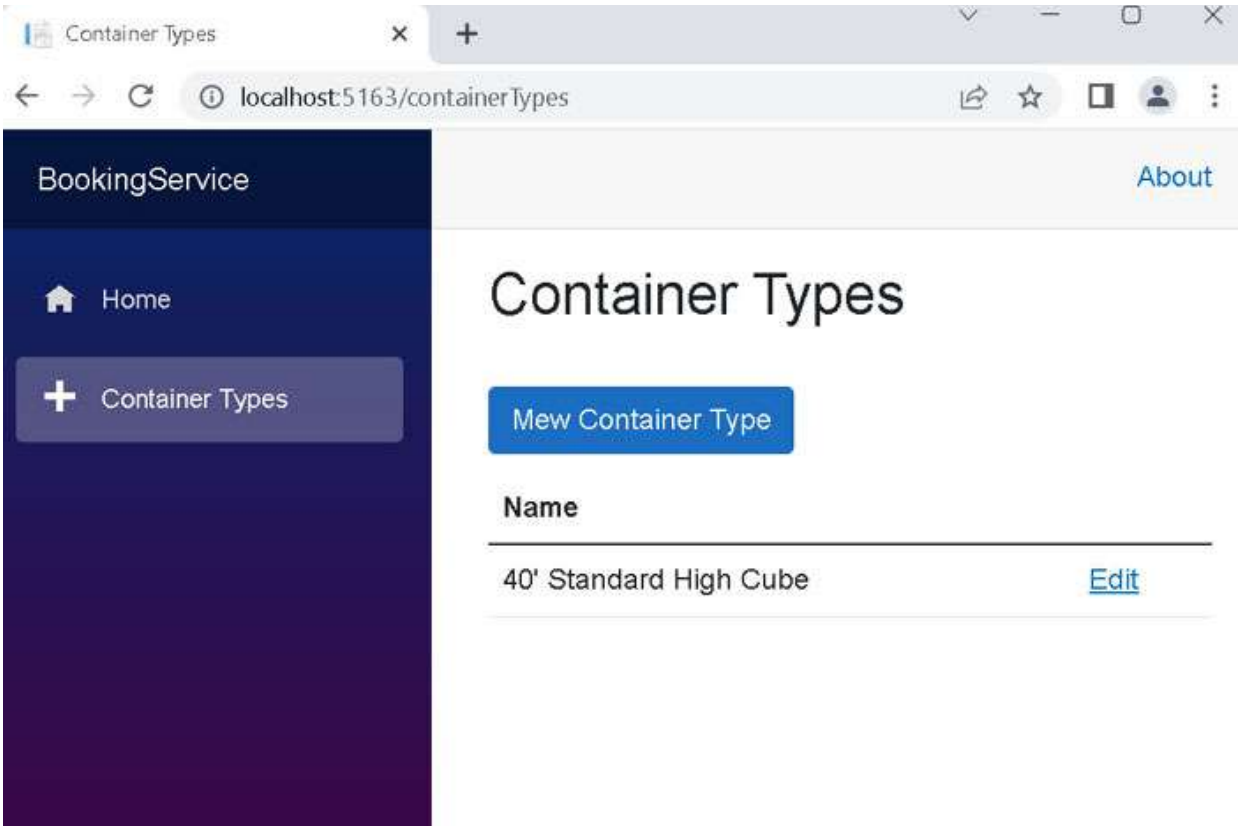


Figure 21.39: *Container type with records*

The same operations done for the Container Type model need to be done for Country, which will be demonstrated in the next section.

Country components

The next step is to create the **`Countries.razor`** component inside the Pages folder. Each Razor component has two main sections: one that is exclusive for HTML, CSS, JS, and Razor code and another section called "code," which contains references to the C# code that includes logical statements associated with the component. After creating the razor corresponding razor component, include the method **`OnInitializedAsync`** method and a private **`countriesList`** property in the code section, as shown in *Figure 21.40*:

```
@code {
    private List<Country>? countriesList;

    protected override async Task OnInitializedAsync()
    {
        var result = await Http.GetAsync("/Country/GetList");
        var stringResult = await result.Content.ReadAsStringAsync();
        countriesList = JsonConvert.DeserializeObject<List<Country>>(stringResult);
    }
}
```

Figure 21.40: *Country code section*

The **`OnInitializedAsync`** method makes an HTTP request to the endpoint we previously created in the **`CountryController`** file in the back end. After making the request, the response is parsed to a list of countries.

The next step is creating the actual HTML that lists the countries populated in the code section. The necessary adaptation in the component should look like the code in *Figure 21.41*:

```
1   @page "/Countries"
2   @using BookingService.Shared
3   @using Newtonsoft.Json
4   @inject HttpClient Http
5
6
7   <PageTitle>Countries</PageTitle>
8
9   <h1>Countries</h1>
10  <br />
11  <div style="position:absolute">
12      <a href="/CountryForm/" class="btn btn-primary">Mew Country</a>
13  </div>
14  <br /><br />
15
16  @if (countriesList == null)
17  {
18      <p><em>Loading...</em></p>
19  }
20  else
21  {
22      <table class="table">
23          <thead>
24              <tr>
25                  <th>Name</th>
26                  <th>Initials</th>
27                  <th>Flag</th>
28                  <th></th>
29              </tr>
30          </thead>
31          <tbody>
32              @foreach (var country in countriesList)
33              {
34                  <tr>
35                      <td>@country.Name</td>
36                      <td>
37
38                          @country.Initials
39
40                      </td>
41                      <td>
42                          <span class="fi fi-@country.Initials.ToLower()"></span>
43
44                      </td>
45                      <td>
46                          <a href="/CountryForm/@country.Id">Edit</a>
47                      </td>
48                  </tr>
49              }
50          </tbody>
51      </table>
52  }
53
```

Figure 21.41: *Country HTML section*

In this component, the route "**`/Countries`**" is specified in the first line using the @page directive, which means this is the address to access the country list page. In line 4, an HttpClient object is being injected, which is a good practice to reuse the same instance multiple times. If you are familiar with Asp.Net MVC with Razor Pages, you will notice that the syntax is identical in terms of creating loops and rendering the dynamic information. In the last

column of the HTML table, there is a link to redirect to the **`CountryForm`**, which will be created in the following.

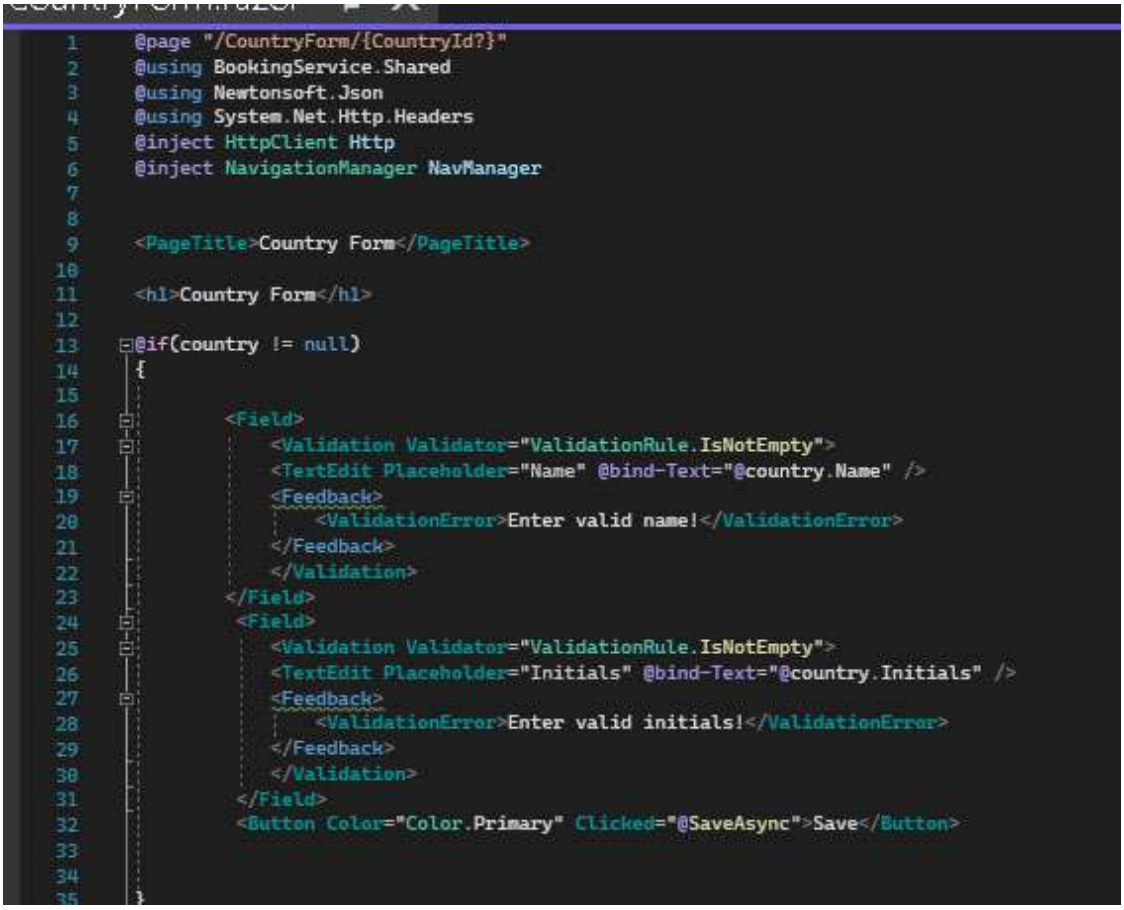
Create a new Razor component called **`CountryForm.razor`**, and include the following content in the code section, as demonstrated in *Figure 21.42*:

```
38  @code {
39      private Country country;
40
41      [Parameter]
42      public string CountryId { get; set; }
43
44      protected override async Task OnInitializedAsync()
45      {
46          if(CountryId != null)
47          {
48              var result = await Http.GetAsync("/Country/Get?id=" + CountryId);
49              var stringResult = await result.Content.ReadAsStringAsync();
50              country = JsonConvert.DeserializeObject<Country>(stringResult);
51          }
52          else
53          {
54              country = new Country();
55          }
56
57      }
58
59      protected async Task SaveAsync()
60      {
61          var myContent = JsonConvert.SerializeObject(country);
62          var buffer = System.Text.Encoding.UTF8.GetBytes(myContent);
63          var stringContent = new StringContent(myContent);
64
65
66          if(country.Id == 0)
67          {
68              await Http.PostAsJsonAsync("/Country/Add", country);
69          }
70          else
71          {
72              await Http.PostAsJsonAsync("/Country/Edit", country);
73          }
74
75          NavManager.NavigateTo("/countries");
76
77      }
78  }
79
```

Figure 21.42: *Country Form code section*

In line 39, a private Country property is specified, representing the model the form will handle. Within this code, there is a logical statement to get from the back-end the corresponding Country if the ID is passed as a parameter. Additionally, the SaveAsync method contains a condition to call different endpoints based on the Add and Edit operations.

As we already have the logical implementation at this stage, we can create the visual part of our component, which is demonstrated in *Figure 21.43*:



```
@page "/CountryForm/{CountryId?}"
@using BookingService.Shared
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject HttpClient Http
@inject NavigationManager NavManager


<PageTitle>Country Form</PageTitle>

<h1>Country Form</h1>

@if(country != null)
{

        <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Name" @bind-Text="@country.Name" />
            <Feedback>
                <ValidationError>Enter valid name!</ValidationError>
            </Feedback>
            </Validation>
        </Field>
         <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Initials" @bind-Text="@country.Initials" />
            <Feedback>
                <ValidationError>Enter valid initials!</ValidationError>
            </Feedback>
            </Validation>
         </Field>
         <Button Color="Color.Primary" Clicked="@SaveAsync">Save</Button>


}
```

Figure 21.43: *Country visual section*

The Razor section of the component uses the Field, Validation, Button, and TextEdit components from the Blazorise framework. In this case, the form warns the user if the Name field's conditions are not satisfied. The Save button calls the **`SaveAsync`** method created previously. The next step is to specify a link to the new pages in the **`NavMenu`** component, which is inside the Shared folder, as shown in *Figure 21.44*:

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="containerTypes">
                <span class="oi oi-plus" aria-hidden="true"></span> Container Types
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Countries">
                <span class="oi oi-plus" aria-hidden="true"></span> Countries
            </NavLink>
        </div>
```

Figure 21.44: *Menu option for Countries*

If you run the application, you will be able to navigate already to the countries page via the lateral menu on the left, as shown in *Figure 21.45*:

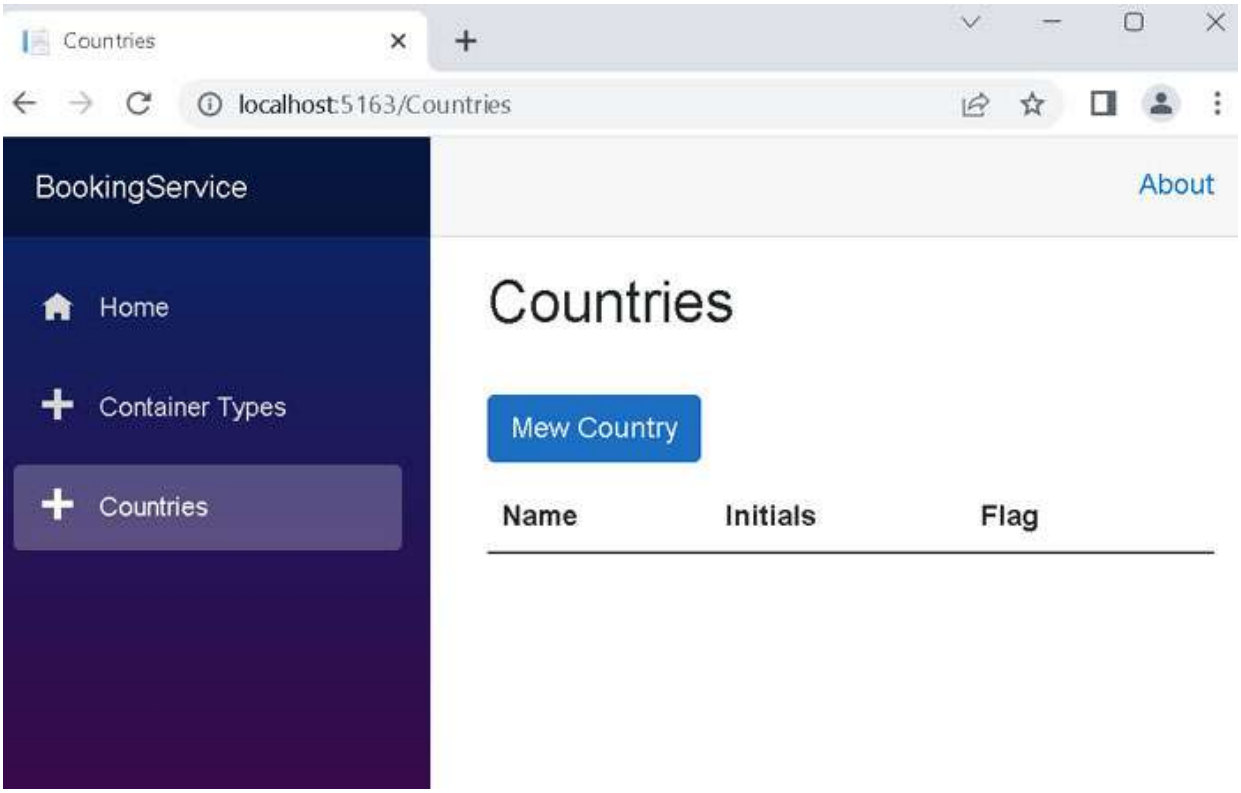


Figure 21.45: *Countries page*

If you click on the **`New Country`** button, you will be redirected to the **`CountryForm`** and will be able to create a new entry in the database, as shown in *Figure 21.46*:

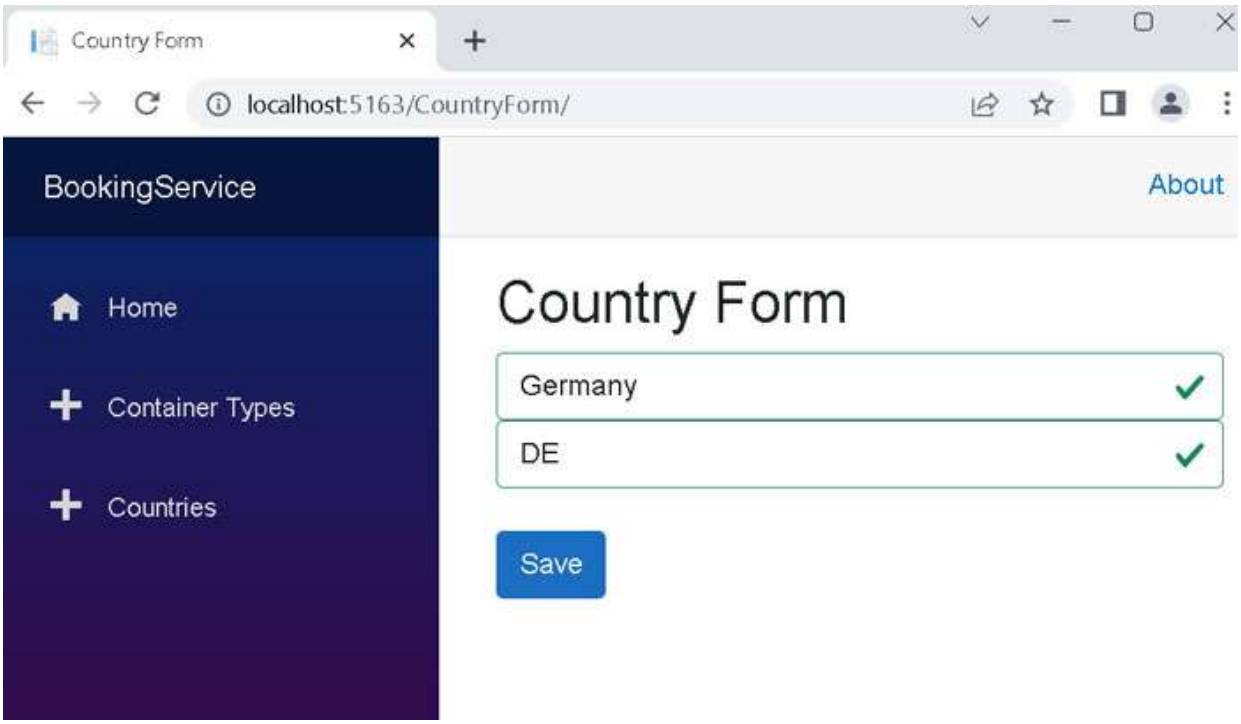


Figure 21.46: *Country form page*

If the validation passes, the fields are marked with green color. After saving the new country, the application redirects to the countries list page again, which is the behavior specified in the **`SaveAsync`** method. With a single record in the database, you will be able to see the entry in the list and will be able to edit the country, as shown in *Figure 21.47*:

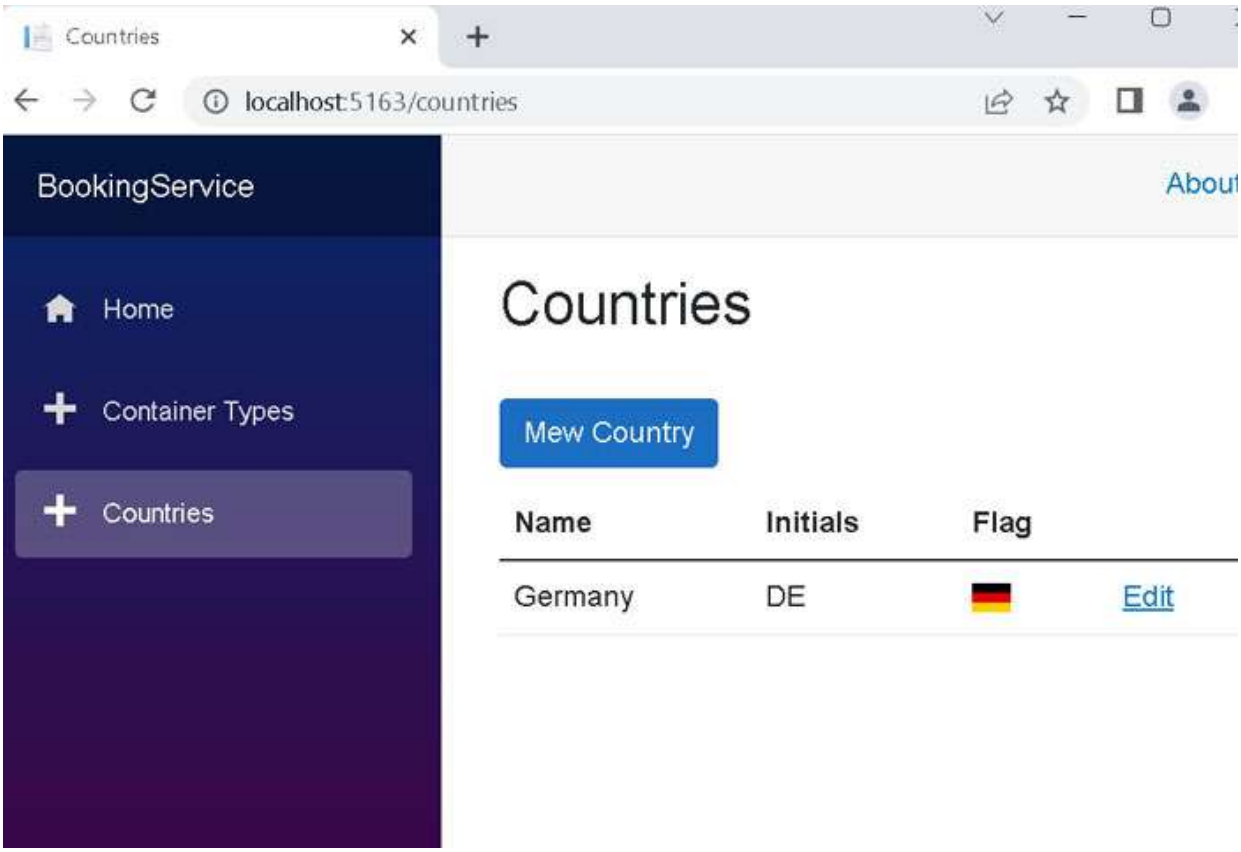


Figure 21.47: *Country with records*

The same operations for the Country model need to be done for the Port, which will be demonstrated in the next section.

Port components

The next step is to create the **`PortList.razor`** component inside the Pages folder. Each Razor component has two main sections: one that is exclusive for HTML, CSS, JS, and Razor code and another section called “code,” which contains references to the C# code that includes logical statements associated with the component. After creating the razor corresponding razor component, include the method **`OnInitializedAsync`** method and a private ports property in the code section, as shown in *Figure 21.48*:

```
@code {
    private List<Port>? ports;

    protected override async Task OnInitializedAsync()
    {
        var result = await Http.GetAsync("/PortInfo/GetList");
        var stringResult = await result.Content.ReadAsStringAsync();
        ports = JsonConvert.DeserializeObject<List<Port>>(stringResult);
    }
}
```

Figure 21.48: *Ports code section*

The OnInitializedAsync method makes an HTTP request to the endpoint that we created previously in the PortInfoController file in the back end. After making the request, the response is parsed to a list of ports.

The next step is to create the actual HTML that will list the ports populated in the code section. The necessary adaptation in the component should look like the code in *Figure 21.49*:

```
1   @page "/PortList"
2   @using BookingService.Shared
3   @using Newtonsoft.Json
4   @inject HttpClient Http
5
6
7   <PageTitle>Port List</PageTitle>
8
9   <h1>Ports</h1>
10  <br />
11  <div style="position:absolute">
12      <a href="/PortForm/" class="btn btn-primary">Mew Port</a>
13  </div>
14  <br /><br />
15
16  @if (ports == null)
17  {
18      <p><em>Loading...</em></p>
19  }
20  else
21  {
22      <table class="table">
23          <thead>
24              <tr>
25                  <th>Name</th>
26                  <th>Country</th>
27                  <th>Flag</th>
28                  <th></th>
29              </tr>
30          </thead>
31          <tbody>
32              @foreach (var port in ports)
33              {
34                  <tr>
35                      <td>@port.Name</td>
36                      <td>
37
38                          @port.Country.Name
39
40                      </td>
41                      <td>
42                          <span class="fi fi-@port.Country.Initials.ToLower()"></span>
43
44                      </td>
45                      <td>
46                          <a href="/PortForm/@port.Id">Edit</a>
47                      </td>
48                  </tr>
49              }
50          </tbody>
51      </table>
52  }
```

Figure 21.49: *Ports HTML section*

In this component, the route "**`/PortList`**" is specified in the first line using the @page directive, which means this is the address to access the port list page. In line 4, an HttpClient object is being injected, which is a good practice to reuse the same instance multiple times. If you are familiar with Asp.Net MVC with Razor Pages, you will notice that the syntax is identical in terms of creating loops and rendering the dynamic information. In the last

column of the HTML table, there is a link to redirect to the PortForm, which will be created in the following.

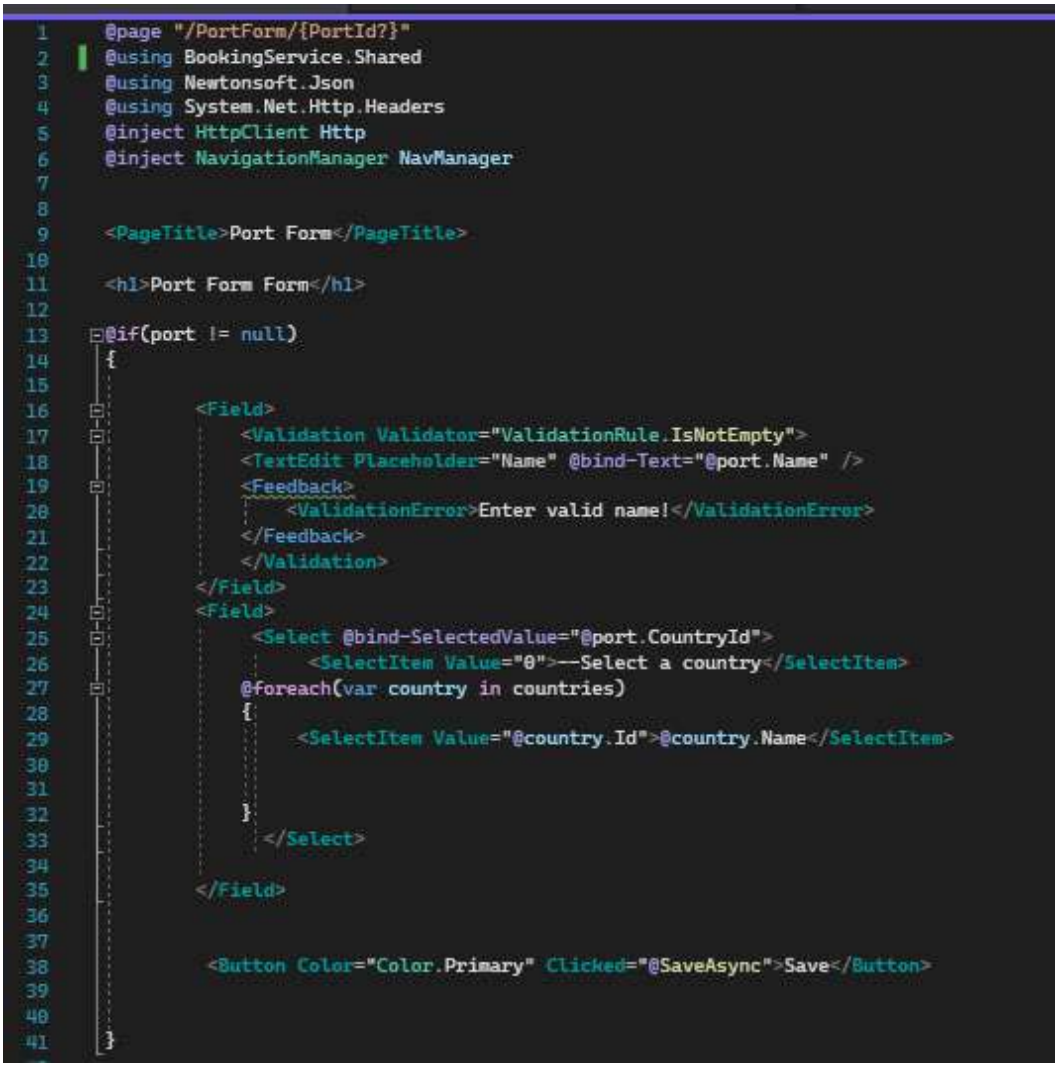
Create a new Razor component called **`PortForm.razor`**, and include the following content in the code section, as demonstrated in *Figure 21.50*:

```
43
44  @code {
45      private Port port;
46      private List<Country> countries;
47
48      [Parameter]
49      public string PortId { get; set; }
50
51      protected override async Task OnInitializedAsync()
52      {
53          var result = await Http.GetAsync("/Country/GetList");
54          var stringResult = await result.Content.ReadAsStringAsync();
55          countries = JsonConvert.DeserializeObject<List<Country>>(stringResult);
56
57          if(PortId != null)
58          {
59              result = await Http.GetAsync("/PortInfo/Get?id=" + PortId);
60              stringResult = await result.Content.ReadAsStringAsync();
61              port = JsonConvert.DeserializeObject<Port>(stringResult);
62          }
63          else
64          {
65              port = new Port();
66          }
67
68      }
69
70      protected async Task SaveAsync()
71      {
72          port.Country = countries.FirstOrDefault(x => x.Id == port.CountryId);
73
74          if(port.Id == 0)
75          {
76              await Http.PostAsJsonAsync("/PortInfo/Add", port);
77          }
78          else
79          {
80              await Http.PostAsJsonAsync("/PortInfo/Edit", port);
81          }
82
83          NavManager.NavigateTo("/portList");
84
85      }
86  }
87
```

Figure 21.50: *Port Form code section*

In line 45, a private Port property is specified, representing the model the form will handle. Within this code, there is a logical statement to get from the back-end the corresponding Port if the ID is passed as a parameter. Additionally, a country list is being populated to be used in the form of a Select element and the SaveAsync method contains a condition to call different endpoints based on the Add and Edit operations.

As we already have the logical implementation at this stage, we can create the visual part of our component, which is demonstrated in *Figure 21.51*:



```
@page "/PortForm/{PortId?}"
@using BookingService.Shared
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject HttpClient Http
@inject NavigationManager NavManager


<PageTitle>Port Form</PageTitle>

<h1>Port Form Form</h1>

@if(port != null)
{

        <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Name" @bind-Text="@port.Name" />
            <Feedback>
                <ValidationError>Enter valid name!</ValidationError>
            </Feedback>
            </Validation>
        </Field>
        <Field>
             <Select @bind-SelectedValue="@port.CountryId">
                 <SelectItem Value="0">--Select a country</SelectItem>
            @foreach(var country in countries)
            {
                <SelectItem Value="@country.Id">@country.Name</SelectItem>


            }
              </Select>

        </Field>


         <Button Color="Color.Primary" Clicked="@SaveAsync">Save</Button>


}
```

Figure 21.51: *Port visual section*

The Razor section of the component uses the Field, Validation, Select, Button, and TextEdit components from the Blazorise framework. In this case, the form warns the user if the conditions for the Name field and other fields are not satisfied. The **`Save`** button calls the **`SaveAsync`** method created previously. The next step is to specify a link to the new pages in the **`NavMenu`** component, which is inside the Shared folder, as shown in *Figure 21.52:*

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="containerTypes">
                <span class="oi oi-plus" aria-hidden="true"></span> Container Types
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Countries">
                <span class="oi oi-plus" aria-hidden="true"></span> Countries
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="portList">
                <span class="oi oi-plus" aria-hidden="true"></span> Ports
            </NavLink>
        </div>

    </nav>
</div>
```

Figure 21.52: *Menu option for Ports*

If you run the application, you will be able to navigate already to the ports page via the lateral menu on the left, as shown in *Figure 21.53*:

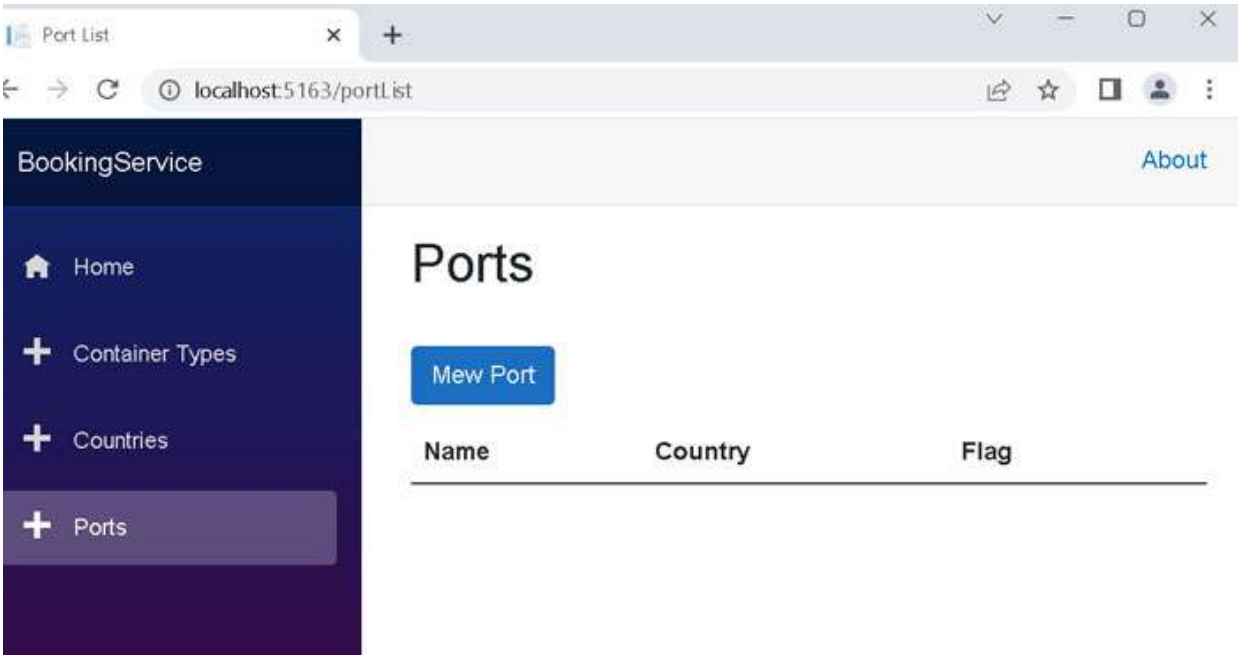


Figure 21.53: *Ports page*

If you click on the New Port button, you will be redirected to the PortForm and will be able to create a new entry in the database, as shown in *Figure*

21.54:

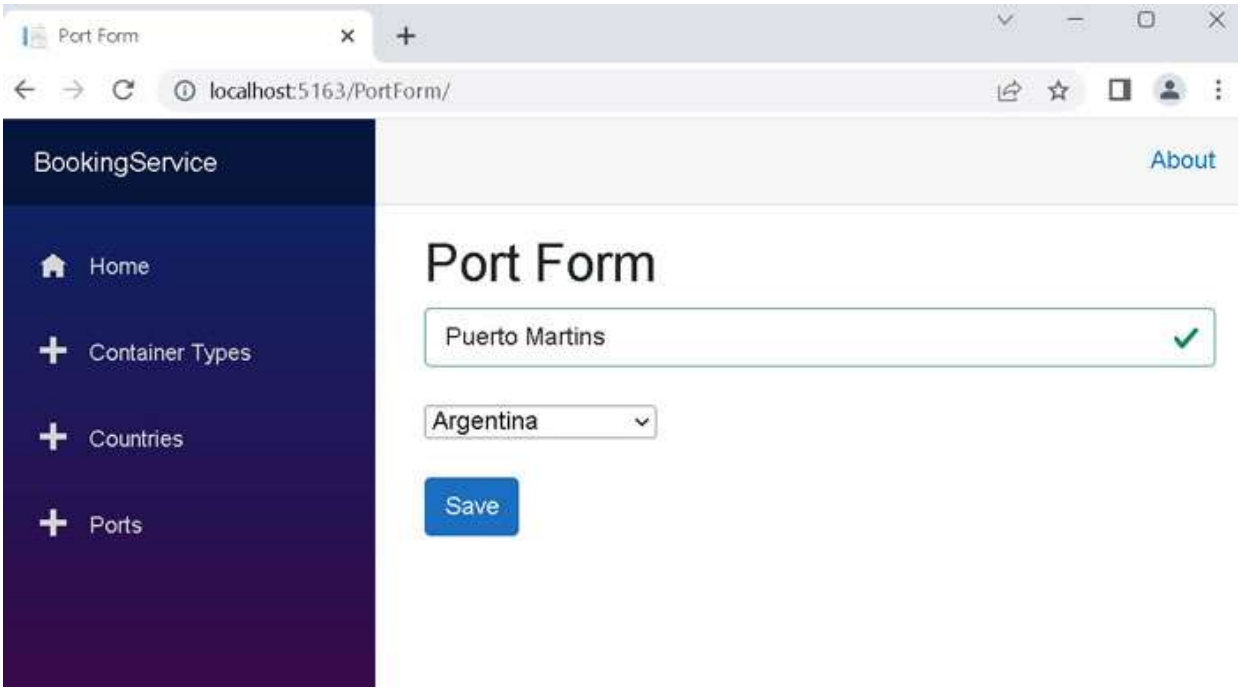


Figure 21.54: *Port form page*

If the validation passes, the fields are marked with green color. After saving the new port, the application redirects to the ports list page again, which is the behavior specified in the **`SaveAsync`** method. With a single record in the database, you will be able to see the entry in the list and will be able to edit the port, as shown in *Figure 21.55*:

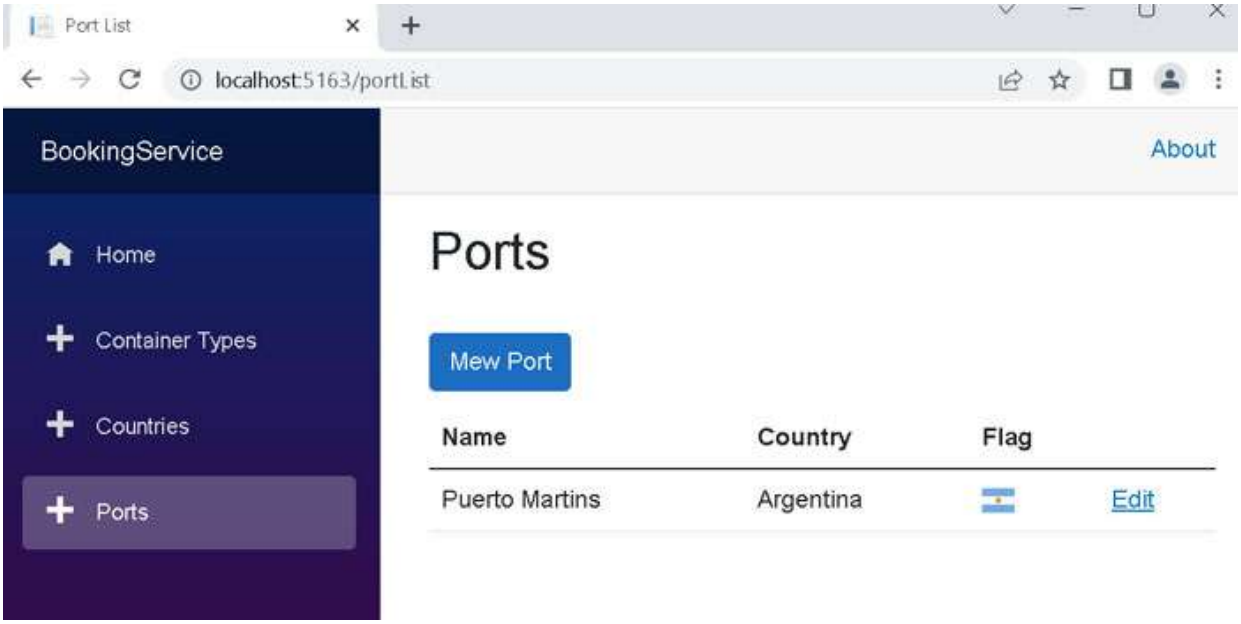


Figure 21.55: *Port with records*

The same operations for the Port model need to be done for the Customer, which will be demonstrated in the next section.

Customer components

The next step is to create the **`CustomerList.razor`** component inside the Pages folder. Each Razor component has two main sections: one that is exclusive for HTML, CSS, JS, and Razor code and another section called "**`code`**," which contains references to the C# code that includes logical statements associated with the component. After creating the razor corresponding razor component, include the method **`OnInitializedAsync`** method and a private customers property in the code section, as shown in *Figure 21.56*:

```
53
54  @code {
55      private List<Customer>? customers;
56
57      protected override async Task OnInitializedAsync()
58      {
59          var result = await Http.GetAsync("/CustomerInfo/GetList");
60          var stringResult = await result.Content.ReadAsStringAsync();
61          customers = JsonConvert.DeserializeObject<List<Customer>>(stringResult);
62      }
63  }
64
```

Figure 21.56: *Customers code section*

The **`OnInitializedAsync`** method makes an HTTP request to the endpoint we previously created in the **`CustomerInfoController`** file in the back end. After making the request, the response is parsed to a list of customers.

The next step is to create the actual HTML that will list the ports populated in the code section. The necessary adaptation in the component should look like the code in *Figure 21.57*:

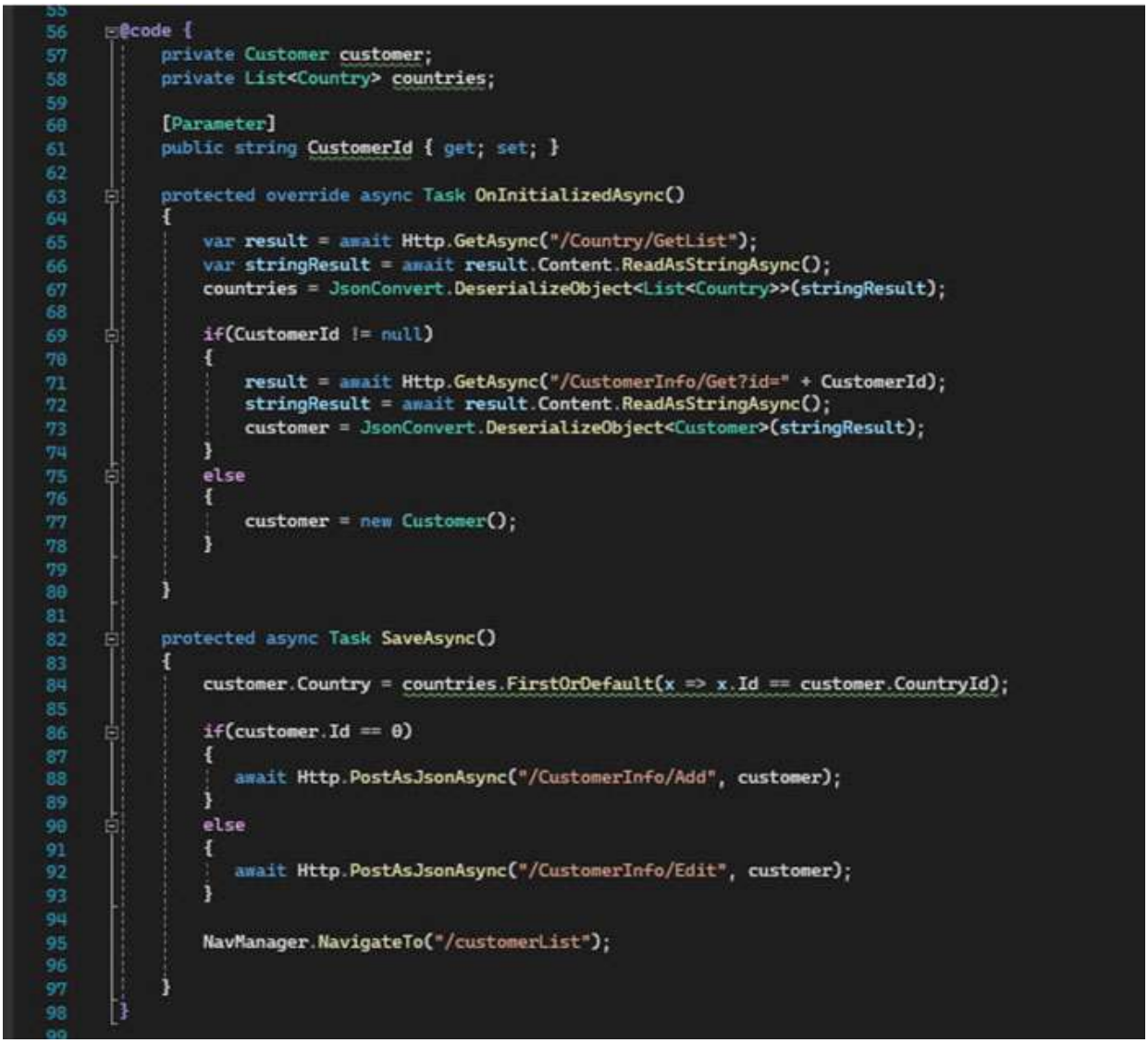
```
1  @page "/CustomerList"
2  @using BookingService.Shared
3  @using Newtonsoft.Json
4  @inject HttpClient Http
5
6
7  <PageTitle>Customer List</PageTitle>
8
9  <h1>Customers</h1>
10 <br />
11 <div style="position:absolute">
12     <a href="/CustomerForm/" class="btn btn-primary">New Customer</a>
13 </div>
14 <br /><br />
15
16 @if (customers == null)
17 {
18     <p><em>Loading...</em></p>
19 }
20 else
21 {
22     <table class="table">
23         <thead>
24             <tr>
25                 <th>Name</th>
26                 <th>Country</th>
27                 <th>Flag</th>
28                 <th></th>
29             </tr>
30         </thead>
31         <tbody>
32             @foreach (var port in customers)
33             {
34                 <tr>
35                     <td>@port.Name</td>
36                     <td>
37
38                         @port.Country.Name
39
40                     </td>
41                     <td>
42                         <span class="fi fi-@port.Country.Initials.ToLower()"></span>
43
44                     </td>
45                     <td>
46                         <a href="/CustomerForm/@port.Id">Edit</a>
47                     </td>
48                 </tr>
49             }
50         </tbody>
51     </table>
52 }
53
```

Figure 21.57: *Customers's HTML section*

In this component, the route "**`/CustomerList`**" is specified in the first line using the @page directive, which means this is the address to access the port list page. In line 4, an HttpClient object is being injected, which is a good practice to reuse the same instance multiple times. If you are familiar with Asp.Net MVC with Razor Pages, you will notice that the syntax is identical in terms of creating loops and rendering the dynamic information. In the last

column of the HTML table, there is a link to redirect to the CustomerForm, which will be created in the following.

Create a new Razor component called **`CustomerForm.razor`**, and include the following content in the code section, as demonstrated in *Figure 21.58*:

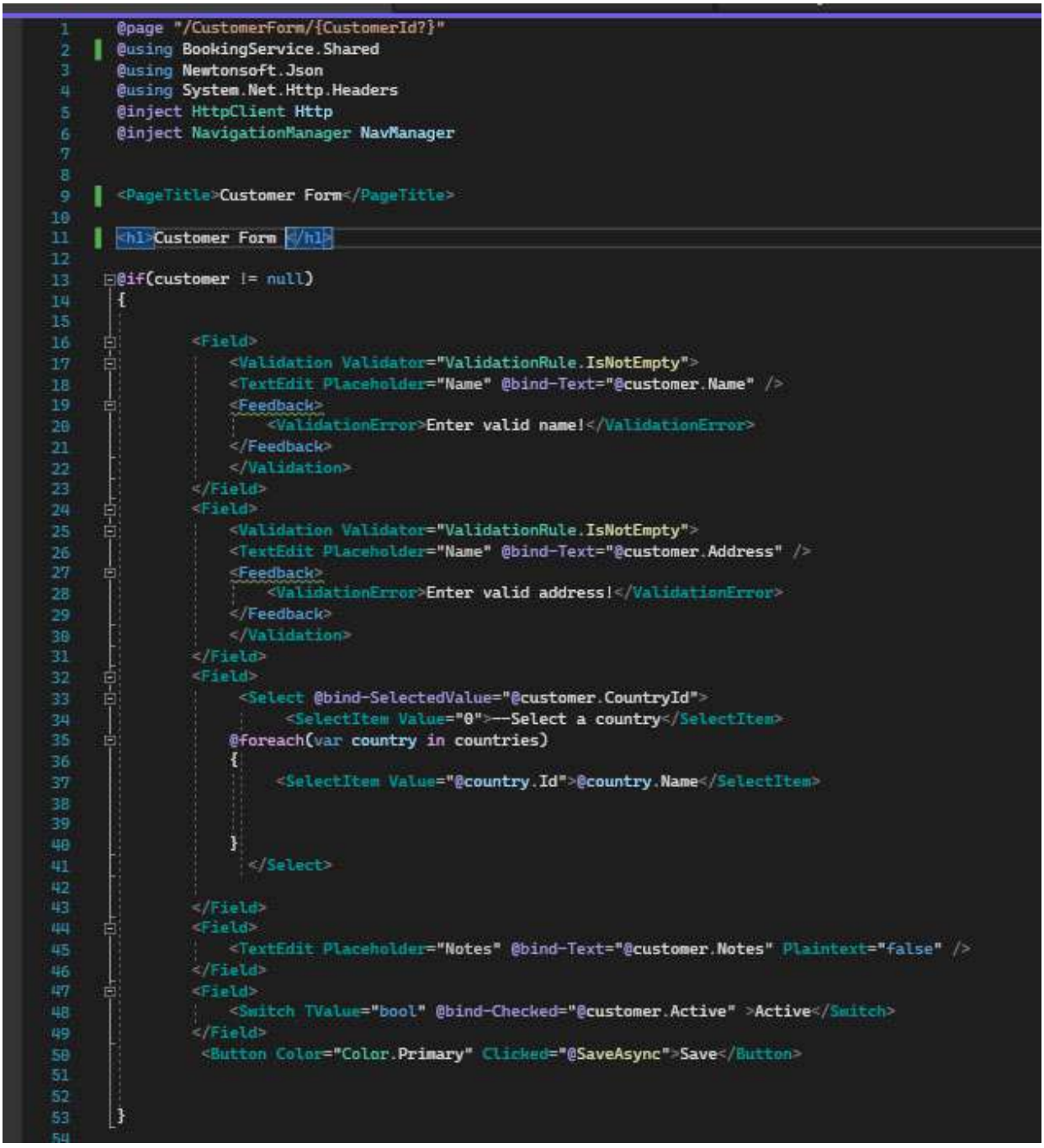


```
@code {
    private Customer customer;
    private List<Country> countries;

    [Parameter]
    public string CustomerId { get; set; }

    protected override async Task OnInitializedAsync()
    {
        var result = await Http.GetAsync("/Country/GetList");
        var stringResult = await result.Content.ReadAsStringAsync();
        countries = JsonConvert.DeserializeObject<List<Country>>(stringResult);

        if(CustomerId != null)
        {
            result = await Http.GetAsync("/CustomerInfo/Get?id=" + CustomerId);
            stringResult = await result.Content.ReadAsStringAsync();
            customer = JsonConvert.DeserializeObject<Customer>(stringResult);
        }
        else
        {
            customer = new Customer();
        }

    }

    protected async Task SaveAsync()
    {
        customer.Country = countries.FirstOrDefault(x => x.Id == customer.CountryId);

        if(customer.Id == 0)
        {
            await Http.PostAsJsonAsync("/CustomerInfo/Add", customer);
        }
        else
        {
            await Http.PostAsJsonAsync("/CustomerInfo/Edit", customer);
        }

        NavManager.NavigateTo("/customerList");

    }
}
```

Figure 21.58: *Customer Form code section*

In line 57, a private Customer property is specified, representing the model the form will handle. Within this code, there is a logical statement to get from the back-end the corresponding Customer if the ID is passed as a parameter. Additionally, a country list is being populated to be used in the form of a Select element and the **`SaveAsync`** method contains a condition to call different endpoints based on the Add and Edit operations.

As we already have the logical implementation at this stage, we can create the visual part of our component, which is demonstrated in *Figure 21.59*:



```
@page "/CustomerForm/{CustomerId?}"
@using BookingService.Shared
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject HttpClient Http
@inject NavigationManager NavManager


<PageTitle>Customer Form</PageTitle>

<h1>Customer Form </h1>

@if(customer != null)
{

        <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Name" @bind-Text="@customer.Name" />
            <Feedback>
                <ValidationError>Enter valid name!</ValidationError>
            </Feedback>
            </Validation>
        </Field>
        <Field>
            <Validation Validator="ValidationRule.IsNotEmpty">
            <TextEdit Placeholder="Name" @bind-Text="@customer.Address" />
            <Feedback>
                <ValidationError>Enter valid address!</ValidationError>
            </Feedback>
            </Validation>
        </Field>
        <Field>
            <Select @bind-SelectedValue="@customer.CountryId">
                <SelectItem Value="0">--Select a country</SelectItem>
            @foreach(var country in countries)
            {
                <SelectItem Value="@country.Id">@country.Name</SelectItem>


            }
            </Select>

        </Field>
        <Field>
            <TextEdit Placeholder="Notes" @bind-Text="@customer.Notes" Plaintext="false" />
        </Field>
        <Field>
            <Switch TValue="bool" @bind-Checked="@customer.Active" >Active</Switch>
        </Field>
         <Button Color="Color.Primary" Clicked="@SaveAsync">Save</Button>


}
```

Figure 21.59: *Customer visual section*

The Razor section of the component uses the **`Field`**, **`Validation`**, **`Select`**, **`Button`**, and **`TextEdit`** components from the Blazorise framework. In this case, the form warns the user if the conditions for the Name field and other

fields are not satisfied. The Save button calls the **`SaveAsync`** method created previously. The next step is to specify a link to the new pages in the **`NavMenu`** component, which is inside the Shared folder, as shown in *Figure 21.60*:

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="containerTypes">
                <span class="oi oi-plus" aria-hidden="true"></span> Container Types
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="Countries">
                <span class="oi oi-plus" aria-hidden="true"></span> Countries
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="portList">
                <span class="oi oi-plus" aria-hidden="true"></span> Ports
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="customerList">
                <span class="oi oi-plus" aria-hidden="true"></span> Customers
            </NavLink>
        </div>

    </nav>
```

Figure 21.60: *Menu option for Customers*

If you run the application, you will be able to navigate already to the customer's page via the lateral menu on the left, as shown in *Figure 21.61*:

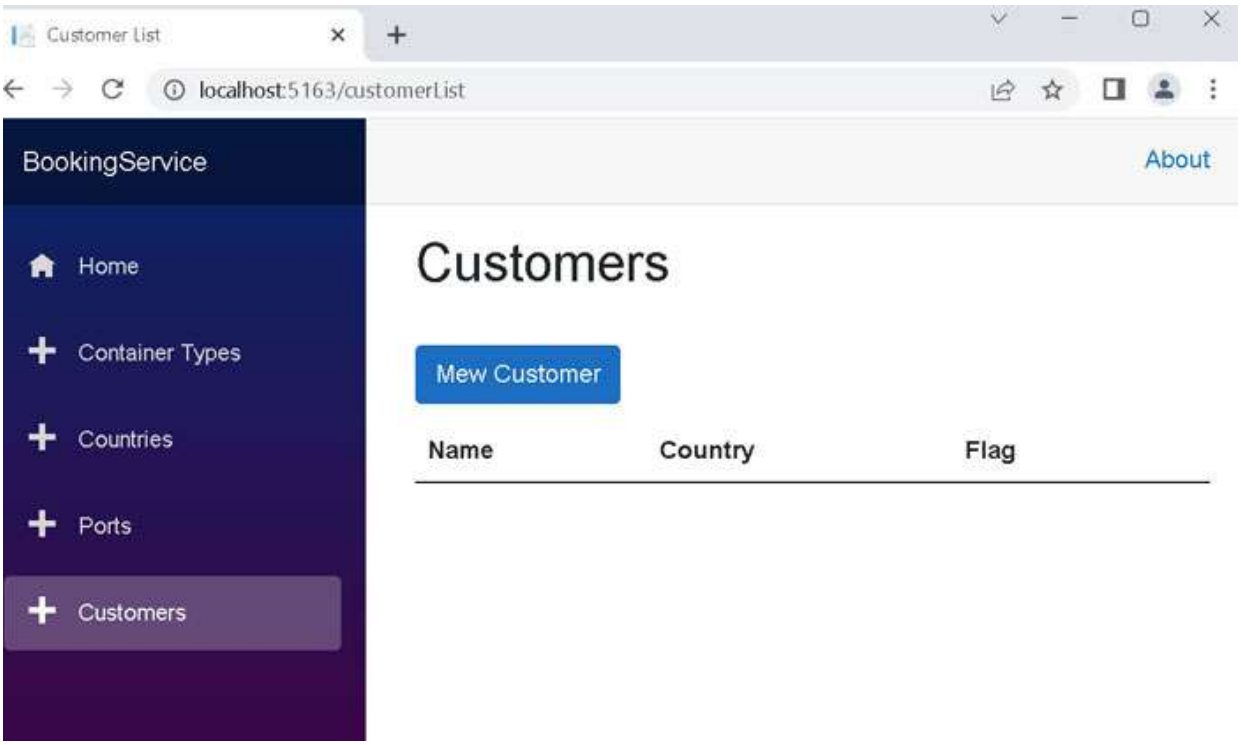


Figure 21.61: *Customers page*

If you click on the **`New Customer`** button, you will be redirected to the **`CustomerForm`** and will be able to create a new entry in the database, as shown in *Figure 21.62*:

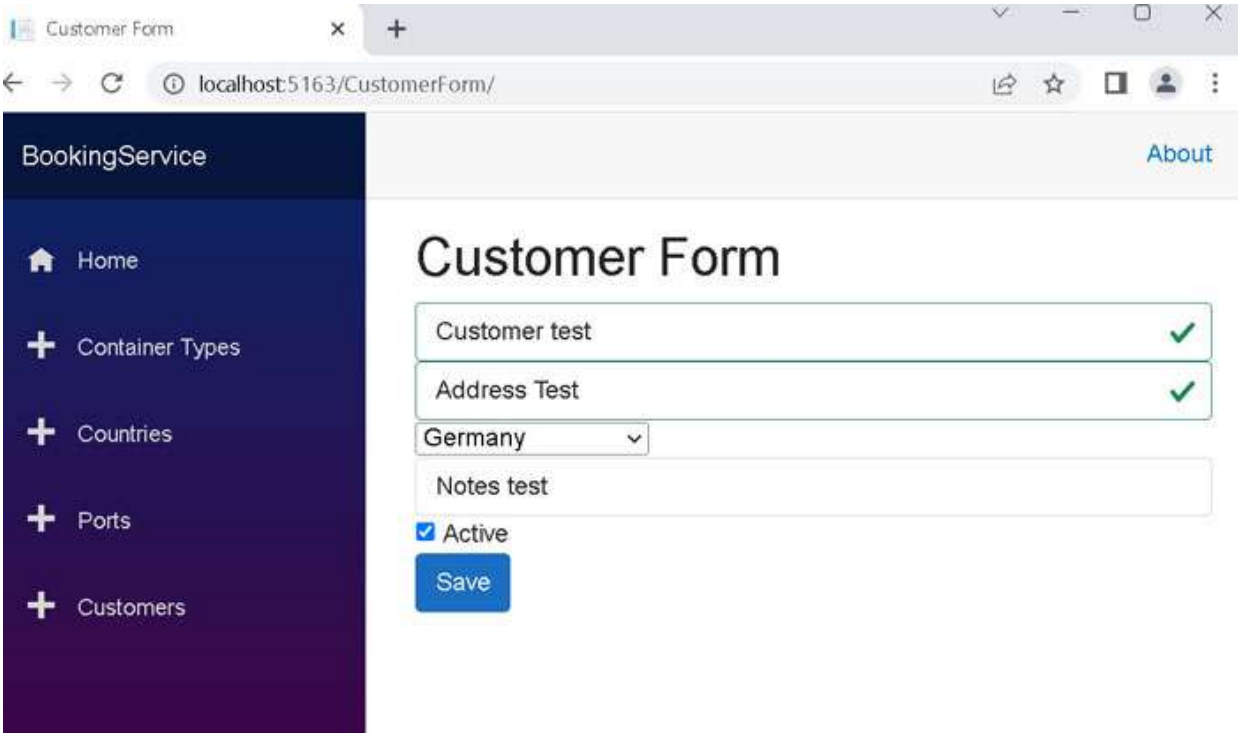


Figure 21.62: *Customer form page*

If the validation passes, the fields are marked with green color. After saving the new port, the application redirects to the ports list page again, which is the behavior specified in the **`SaveAsync`** method. With a single record in the database, you will be able to see the entry in the list and will be able to edit the customer, as shown in *Figure 21.63*:

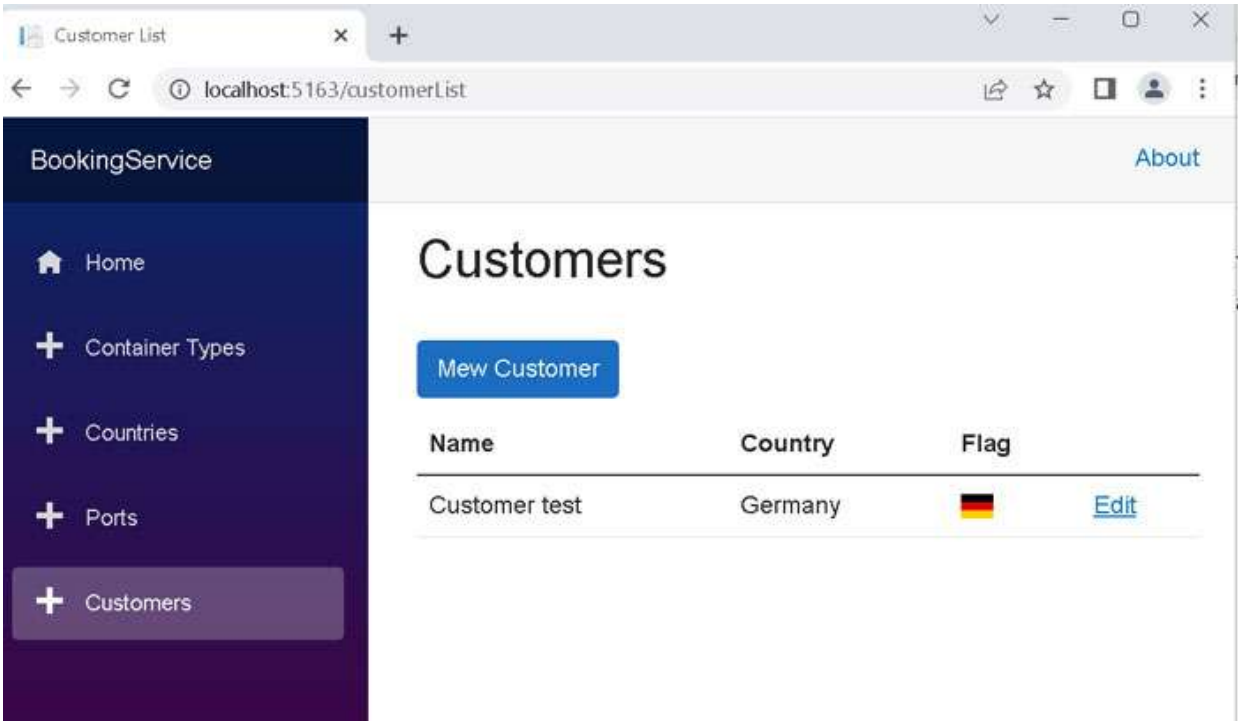


Figure 21.63: *Customer with records*

The same operations done for the Customer model need to be done for Booking, which will be demonstrated in the next section.

Booking components

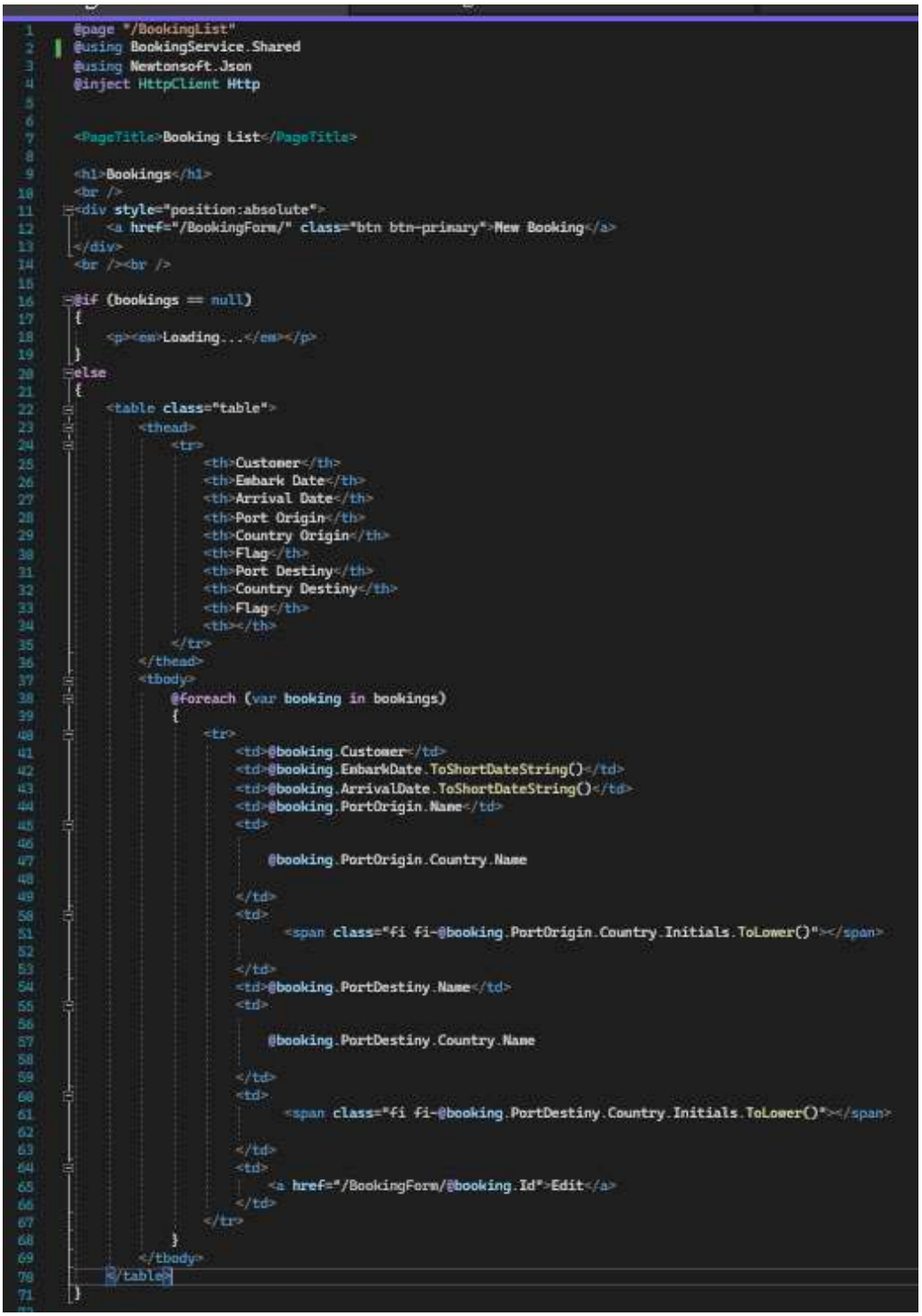
The next step is to create the **`BookingList.razor`** component inside the Pages folder. After creating the razor corresponding razor component, include the method **`OnInitializedAsync`** method and a private bookings property in the code section, as shown in *Figure 21.64*:

```
72
73  @code {
74      private List<Booking>? bookings;
75
76      protected override async Task OnInitializedAsync()
77      {
78          var result = await Http.GetAsync("/BookingInfo/GetList");
79          var stringResult = await result.Content.ReadAsStringAsync();
80          bookings = JsonConvert.DeserializeObject<List<Booking>>(stringResult);
81      }
82  }
83
```

Figure 21.64: *Bookings code section*

The **`OnInitializedAsync`** method makes an HTTP request to the endpoint we created previously in the **`BookingsController`** file in the back end. After making the request, the response is parsed to a list of bookings.

The next step is to create the actual HTML that will list the ports populated in the code section. The necessary adaptation in the component should look like the code in *Figure 21.65*:



```
1   @page "/BookingList"
2   @using BookingService.Shared
3   @using Newtonsoft.Json
4   @inject HttpClient Http
5
6
7   <PageTitle>Booking List</PageTitle>
8
9   <h1>Bookings</h1>
10  <br />
11  <div style="position:absolute">
12      <a href="/BookingForm/" class="btn btn-primary">New Booking</a>
13  </div>
14  <br /><br />
15
16  @if (bookings == null)
17  {
18      <p><em>Loading...</em></p>
19  }
20  else
21  {
22      <table class="table">
23          <thead>
24              <tr>
25                  <th>Customer</th>
26                  <th>Embark Date</th>
27                  <th>Arrival Date</th>
28                  <th>Port Origin</th>
29                  <th>Country Origin</th>
30                  <th>Flag</th>
31                  <th>Port Destiny</th>
32                  <th>Country Destiny</th>
33                  <th>Flag</th>
34                  <th></th>
35              </tr>
36          </thead>
37          <tbody>
38              @foreach (var booking in bookings)
39              {
40                  <tr>
41                      <td>@booking.Customer</td>
42                      <td>@booking.EmbarkDate.ToShortDateString()</td>
43                      <td>@booking.ArrivalDate.ToShortDateString()</td>
44                      <td>@booking.PortOrigin.Name</td>
45                      <td>
46
47                          @booking.PortOrigin.Country.Name
48
49                      </td>
50                      <td>
51                          <span class="fi fi-@booking.PortOrigin.Country.Initials.ToLower()"></span>
52
53                      </td>
54                      <td>@booking.PortDestiny.Name</td>
55                      <td>
56
57                          @booking.PortDestiny.Country.Name
58
59                      </td>
60                      <td>
61                          <span class="fi fi-@booking.PortDestiny.Country.Initials.ToLower()"></span>
62
63                      </td>
64                      <td>
65                          <a href="/BookingForm/@booking.Id">Edit</a>
66                      </td>
67                  </tr>
68              }
69          </tbody>
70      </table>
71  }
```

Figure 21.65: *Bookings HTML section*

In this component, the route "**`/BookingList`**" is specified in the first line using the @page directive, which means this is the address to access the port list page. In line 4, an **`HttpClient`** object is being injected, which is a good

practice to reuse the same instance multiple times. If you are familiar with Asp.Net MVC with Razor Pages, you will notice that the syntax is identical in terms of creating loops and rendering the dynamic information. In the last column of the HTML table, there is a link to redirect to the **`BookingForm`**, which will be created in the following.

Create a new Razor component called **`BookingForm.razor`**, and include the following content in the code section, as demonstrated in *Figure 21.66*:

```
@code {
    private Booking bookingRecord;
    private List<Port> ports;
    private List<ContainerType> containerTypes;
    private List<Customer> customers;

    [Parameter]
    public string BookingId { get; set; }

    protected override async Task OnInitializedAsync()
    {
        var result = await Http.GetAsync("/PortInfo/GetList");
        var stringResult = await result.Content.ReadAsStringAsync();
        ports = JsonConvert.DeserializeObject<List<Port>>(stringResult);

        result = await Http.GetAsync("/ContainerType/GetList");
        stringResult = await result.Content.ReadAsStringAsync();
        containerTypes = JsonConvert.DeserializeObject<List<ContainerType>>(stringResult);

        result = await Http.GetAsync("/CustomerInfo/GetList");
        stringResult = await result.Content.ReadAsStringAsync();
        customers = JsonConvert.DeserializeObject<List<Customer>>(stringResult);

        if(BookingId != null)
        {
            result = await Http.GetAsync("/BookingInfo/Get?id=" + BookingId);
            stringResult = await result.Content.ReadAsStringAsync();
            bookingRecord = JsonConvert.DeserializeObject<Booking>(stringResult);
        }
        else
        {
            bookingRecord = new Booking();
        }

    }

    protected async Task SaveAsync()
    {
        bookingRecord.Customer = customers.FirstOrDefault(x => x.Id == bookingRecord.CustomerId);
        bookingRecord.ContainerType = containerTypes.FirstOrDefault(x=> x.Id == bookingRecord.ContainerTypeId);
        bookingRecord.PortOrigin = ports.FirstOrDefault(x => x.Id == bookingRecord.PortOriginId);
        bookingRecord.PortDestiny = ports.FirstOrDefault(x => x.Id == bookingRecord.PortDestinyId);



        if(bookingRecord.Id == 0)
        {
           await Http.PostAsJsonAsync("/BookingInfo/Add", bookingRecord);
        }
        else
        {
           await Http.PostAsJsonAsync("/BookingInfo/Edit", bookingRecord);
        }

        NavManager.NavigateTo("/bookingList");

    }
}
```

Figure 21.66: *Booking Form code section*

In line 79, a private Booking property is specified, representing the model the form will handle. Within this code, there is a logical statement to get from the back-end the corresponding Booking if the ID is passed as a parameter. Additionally, two port lists are being populated to be used in the form of Select elements. The **`SaveAsync`** method contains a condition to call different endpoints based on the Add and Edit operations.

As we already have the logical implementation at this stage, we can create the visual part of our component, which is demonstrated in *Figure 21.67*:

```
@page "/BookingForm/{BookingId?}"
@using BookingService.Shared
@using Newtonsoft.Json
@using System.Net.Http.Headers
@inject HttpClient Http
@inject NavigationManager NavManager


<PageTitle>Booking Form</PageTitle>

<h1>Booking Form</h1>

@if(bookingRecord != null)
{
        <Field>
            <Select @bind-SelectedValue="@bookingRecord.CustomerId">
                <SelectItem Value="0">--Select a Customer</SelectItem>
            @foreach(var customer in customers)
            {
                <SelectItem Value="@customer.Id">@customer.Name</SelectItem>


            }
              </Select>

        </Field>
         <Field>
            <Select @bind-SelectedValue="@bookingRecord.ContainerTypeId">
                <SelectItem Value="0">--Select a Container Type</SelectItem>
            @foreach(var containerType in containerTypes)
            {
                <SelectItem Value="@containerType.Id">@containerType.Name</SelectItem>


            }
              </Select>

        </Field>
        <Field>

            <DateEdit TValue="DateTime" @bind-Date="@bookingRecord.EmbarkDate" Placeholder="Embark Date" />
        </Field>
        <Field>
            <DateEdit TValue="DateTime" @bind-Date="@bookingRecord.ArrivalDate" Placeholder="Arrival Date" />
        </Field>
        <Field>
            <Select @bind-SelectedValue="@bookingRecord.PortOriginId">
                <SelectItem Value="0">--Select a Port Origin</SelectItem>
            @foreach(var port in ports)
            {
                <SelectItem Value="@port.Id">@port.Name</SelectItem>


            }
              </Select>

        </Field>
        <Field>
            <Select @bind-SelectedValue="@bookingRecord.PortDestinyId">
                <SelectItem Value="0">--Select a Port Destiny</SelectItem>
            @foreach(var port in ports)
            {
                <SelectItem Value="@port.Id">@port.Name</SelectItem>


            }
              </Select>

        </Field>

         <br />
         <Button Color="Color.Primary" Clicked="@SaveAsync">Save</Button>

}
```

Figure 21.67: *Booking visual section*

The Razor section of the component uses the **`Field`**, **`Validation`**, **`Select`**, **`Button`**, and **`TextEdit`** components from the Blazorise framework. In this

case, the form warns the user if the conditions for the Name field and other fields are not satisfied. The Save button calls the **`SaveAsync`** method created previously. The next step is to specify a link to the new pages in the **`NavMenu`** component, which is inside the Shared folder, as shown in *Figure 21.68*:

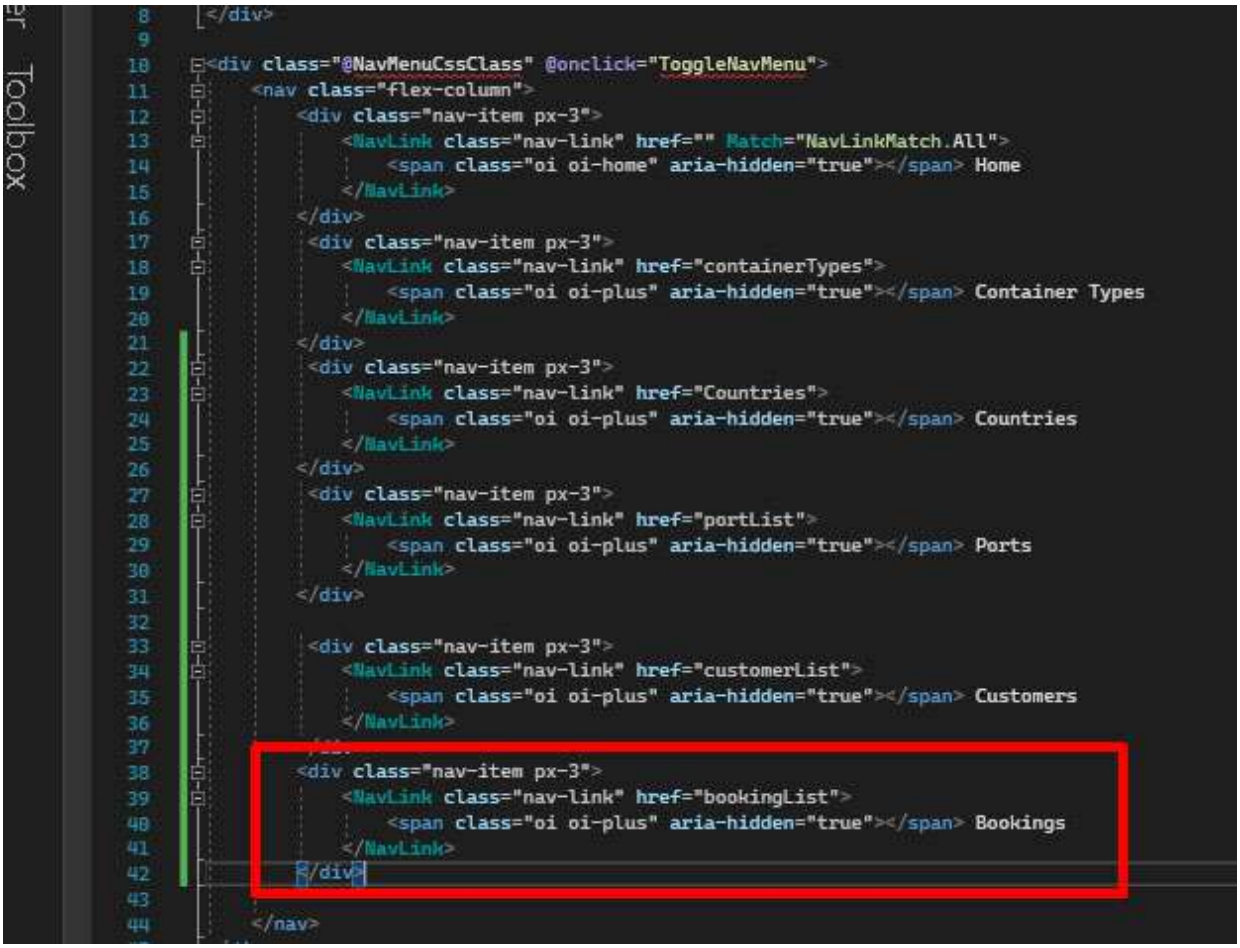


Figure 21.68: *Menu option for Bookings*

If you run the application, you will be able to navigate already to the bookings page via the lateral menu on the left, as shown in *Figure 21.69*:

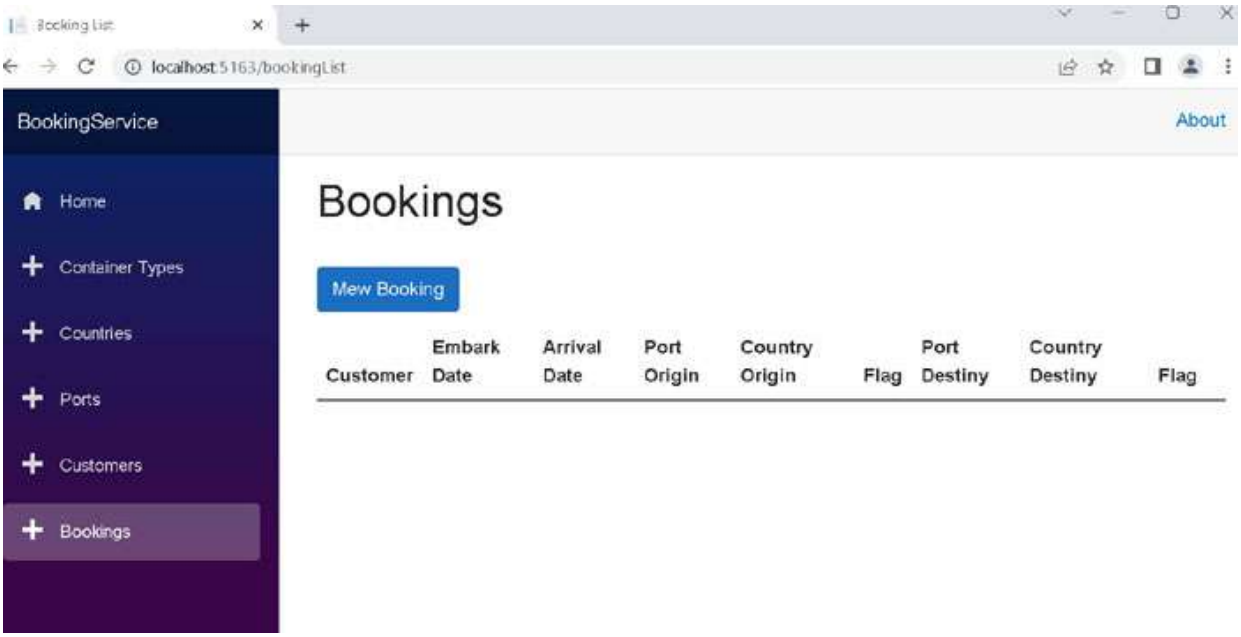


Figure 21.69: *Bookings page*

If you click on the New Booking button, you will be redirected to the **`BookingForm`** and will be able to create a new entry in the database, as shown in *Figure 21.70*:

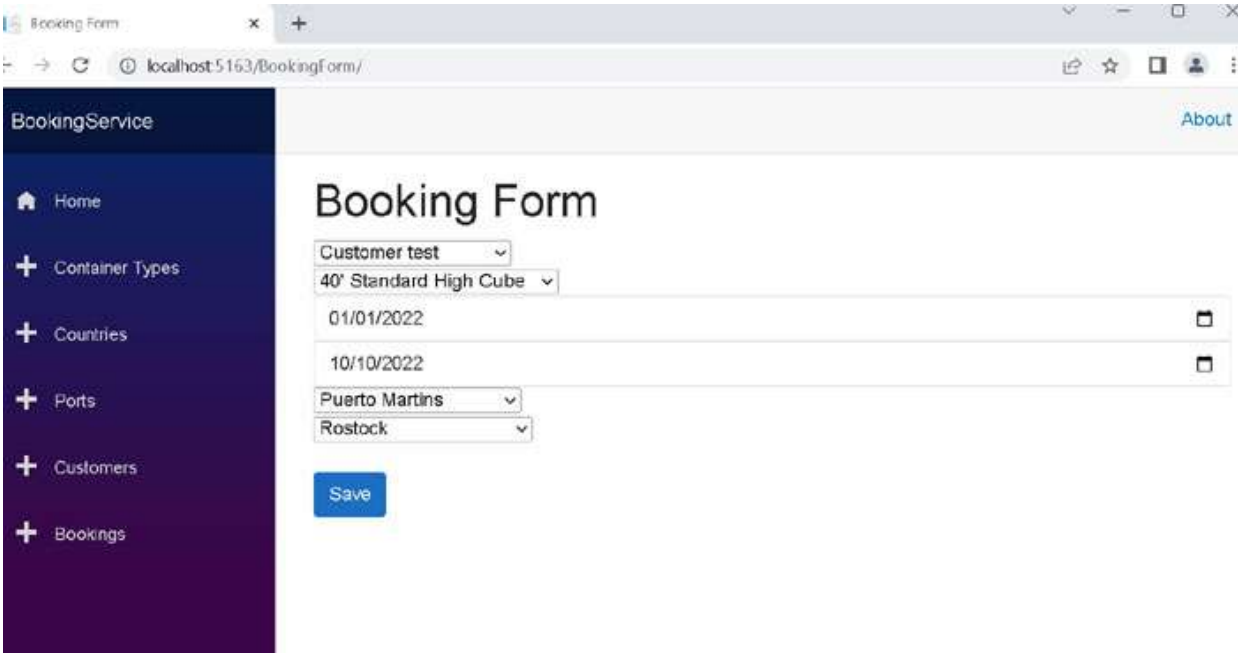


Figure 21.70: *Booking form page*

If the validation passes, the fields are marked with green color. After saving the new port, the application redirects to the bookings list page again, as it is

the behavior specified in the **`SaveAsync`** method. With a single record in the database, you will be able to see the entry in the list and will be able to edit the booking, as shown in *Figure 21.71*:

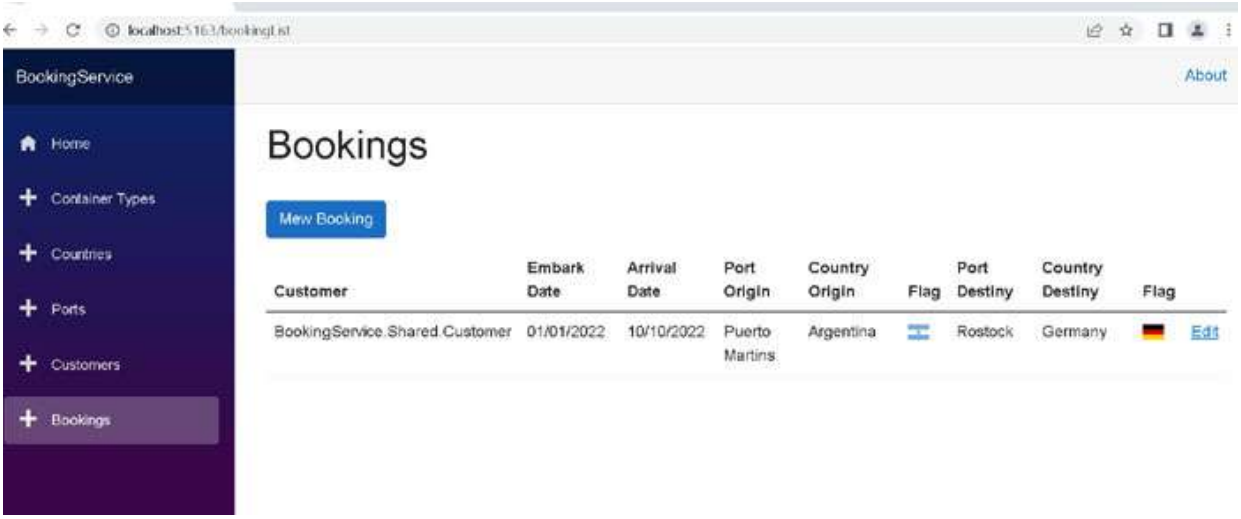


Figure 21.71: *Customer with records*

As you have the most basic operations for the Booking Service working, we advise you to apply to the same project what you have learned in this book about Authentication and Unit Testing.

Conclusion

In this chapter, you learned how to create a basic application for a logistic company and all the layers for a complete system, including the back-end and front-end. The application's creation involved many things you learned with this book, such as object-oriented paradigm, SOLID principles, Entity Framework, Asp.Net Core Web API, Blazor Server, Blazor WebAssembly, Razor components, and much more.

I hope this book helped you to be a better software developer and contributed to your capabilities of building scalable and sustainable enterprise applications using the .NET platform and the C# language.

Points to remember

- Routes are specified in the @page directive for Razor components.

- Blazorise is an open-source UI framework to be used with Blazor applications.
- It is possible to inject objects into Razor components.
- The Asp.Net Core Hosted option needs to be used if there is a need to combine. Front-end and back-end development for Blazor applications.

Questions

1. Implement unit tests for the Booking Service project using the knowledge you gained with this book.
2. Develop authentication capabilities for the application.
3. Refactor the Booking Service back-end to use dependency injection within Controllers for the Business Logic objects.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X