Learn ASP.NET Core 3

Second Edition

Develop modern web applications with ASP.NET Core 3, Visual Studio 2019, and Azure

Kenneth Yamikani Fukizi, Jason De Oliveira and Michel Bruchet

WOW! eBook
www.wowebook.org

Packt
www.packt.com

Learn ASP.NET Core 3
Second Edition

Develop modern web applications with ASP.NET Core 3, Visual Studio 2019, and Azure

Kenneth Yamikani Fukizi
Jason De Oliveira
Michel Bruchet

Packt

BIRMINGHAM - MUMBAI

WOW! eBook
www.wowebook.org

Learn ASP.NET Core 3
Second Edition

Copyright © 2019 Packt Publishing

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

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

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

Commissioning Editor: Richa Tripathi
Acquisition Editor: Karan Gupta
Content Development Editor: Pathikrit Roy
Senior Editor: Rohit Singh
Technical Editor: Gaurav Gala
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Arvindkumar Gupta

First published: December 2017
Second edition: December 2019

Production reference: 1261219

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78961-013-0

www.packt.com

WOW! eBook
www.wowebook.org

To my beautiful wife, Eva, my lovely daughters, Amara and Annelise, and my handsome son, Josh:

Thank you, guys, for enduring my long extra working hours, to the point that Annelise decided that she no longer wants to become a software programmer anymore because she doesn't like the fact that I'm always working!

To my parents, Kenneth Jester Fukizi and Emily Mchepa:

You did a good job.

- Kenneth Yamikani Fukizi

WOW! eBook
www.wowebook.org

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

WOW! eBook
www.wowebook.org

Contributors

About the authors

Kenneth Yamikani Fukizi is a software engineer, solutions architect, and consultant with more than 14 years of experience.

He is passionate about programming and web platforms. His experience includes working as a software engineering contractor/consultant on various projects for clients based in South Africa, Australia, the U.S.A, and Canada.

Kenneth is based in Cape Town and is the founder of the AfrikanCoder™ project, on which he works on a part-time basis. Kenneth is a Microsoft Certified Trainer®, Microsoft Certified Solutions Developer®, and has other technical qualifications. He holds a bachelor's degree in computer science and a master's degree in finance and is currently pursuing a PhD in computer science.

A special thanks to the team of editors at Packt whose professional help and guidance is second to none; I really enjoyed working with the team.

I would like to acknowledge the authors of the first edition of this book, Jason De Oliveira and Michel Bruchet, for the tremendous and timeless job they did in the first edition. This book would have been a much bigger task without their initial contribution.

WOW! eBook
www.wowebook.org

Jason De Oliveira works as a CTO for MEGA International, a software company in Paris. He is an experienced manager and senior solutions architect, with high-level skills in software architecture and enterprise architecture.

He loves sharing his knowledge and experience via blogs, conferences, books, articles, courses, and the coaching of co-workers in his company. He has also worked on many great technical books in English and French. He frequently collaborates with Microsoft and can often be found at the **Microsoft Technology Center** (**MTC**) in Paris.

Microsoft has awarded him for over 6 years with the Microsoft Most Valuable Professional in C#/.NET award for his numerous contributions to the Microsoft community.

> *I would like to thank my lovely wife, Orianne, and my beautiful daughters, Julia and Léonie, for supporting me in my work and for accepting long days and short nights during the week, and, sometimes, even during the weekend. My life would not be the same without them!*

Michel Bruchet works as an application architect for MEGA International, a software company in Paris. He has over 20 years of experience as a senior architect, working on complex projects in IT and development departments.

Michel has published several publications on the internet (to be found on SlideShare, LinkedIn, and more). He has worked for big companies in France, including Sanofi, Pierre et Vacances – Center Parcs, Banque de France, BPCE, and BNP.

He is also the main driving force and mastermind behind Ingenius Solution, which provides efficient e-business solutions to customers around the world.

> *I would like to thank my family for accepting that I had to work hard and, sometimes, late into the night in my spare time to write this book!*

WOW! eBook
www.wowebook.org

About the reviewer

Alvin Ashcraft is a developer living near Philadelphia. He has spent his 23-year career building software with C#, Visual Studio, WPF, ASP.NET, and more. He has been awarded the Microsoft MVP title on nine occasions. You can read his daily links for .NET developers on his blog, Morning Dew. He works as a principal software engineer for Allscripts, building healthcare software. He has previously been employed by software companies, including Oracle. He has reviewed other titles for Packt Publishing, such as *Mastering ASP.NET Core 2.0*, *Mastering Entity Framework Core 2.0*, and *Learning ASP.NET Core 2.0*.

I would like to thank my wonderful wife, Stelene, and our three amazing daughters for their support. They were very understanding when I was reading and reviewing these chapters in the evenings and at weekends to help deliver a useful, high-quality book for .NET developers.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

WOW! eBook
www.wowebook.org

Table of Contents

Preface 1

Section 1: Section 1: Introduction and Environment Setup

Chapter 1: What Is ASP.NET Core 3? 13
- **The history of ASP.NET** 14
- **ASP.NET Core 3 features** 16
- **What is new specifically to ASP.NET Core 3?** 17
- **Cross-platform support** 19
- **Microservice architecture** 21
  - Working with containers 22
- **Performance and scalability** 22
- **Technology restrictions** 23
  - Common technologies not directly found in ASP.NET Core and .NET Core 24
- **When to choose ASP.NET Core 3** 24
- **Summary** 25

Chapter 2: Setting Up the Environment 26
- **Visual Studio 2019 as a development environment** 27
- **How to install Visual Studio 2019 Community Edition** 28
  - First steps with Visual Studio 2019 32
  - Creating your first ASP.NET Core 3 application in Visual Studio 2019 36
  - Creating your first ASP.NET Core 3 application via the command line 41
- **Basic debugging with Visual Studio 2019** 43
  - Breakpoints 45
  - Call stack 46
  - Autos, Locals, and Watch Panes 47
- **Visual Studio Code as a development environment** 47
- **How to install Visual Studio Code on Linux** 49
  - Creating your first ASP.NET Core 3 application in Visual Studio Code 52
  - Creating your first ASP.NET Core 3 application in Linux 54
- **Introduction to the C# Interactive and LINQPad tools** 56
- **Summary** 58

Chapter 3: Continuous Integration Pipeline in Azure DevOps 59
- **Technical requirements** 60
- **CI, CD, and build and release pipelines** 60
  - Using Azure DevOps for CI and CD 61
  - Creating a free Azure DevOps subscription and your first Azure DevOps project 62

WOW! eBook
www.wowebook.org

Organizing your work via work items 64
Understanding the scrum process 66
Using Git as a VCS 71
Using feature branches 77
Merging changes and resolving conflicts 79
Creating an Azure DevOps build pipeline 83
Creating an Azure DevOps release pipeline 87
Summary 88

Section 2: Section 2: A Practical Demonstration of ASP.NET Core 3

Chapter 4: Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1 90
Preview of the Tic-Tac-Toe demo application 91
Building the Tic-Tac-Toe game 92
Conceiving and implementing your first Tic-Tac-Toe feature 93
Targeting different .NET Core versions in the .csproj files of your projects 97
Using the Microsoft.AspNetCore.App metapackage 98
Introduction to the default ASP.NET Core 3 classes 99
ASP.NET Core 3 start up classes 100
Working with the Program class 100
Working with .NET Generic Host instead of WebHostBuilder 102
Working with the Startup class 103
Preparing the basic project structure 105
Creating the Tic-Tac-Toe home page 107
Giving your web pages a more modern look by using NPM and layout pages 112
Updating the layout page 115
Creating the Tic-Tac-Toe user registration page 119
Creating the Tic-Tac-Toe user service 122
Using DI to encourage loose coupling 122
Creating the user service 123
Creating a basic communication middleware for the Tic-Tac-Toe application 127
Working with middleware 127
Creating the communication middleware 130
Working with static files 132
Using routing, URL redirection, and URL rewriting 134
Endpoint routing for ASP.NET Core 3 137
Adding error handling to the Tic-Tac-Toe application 138
Summary 144

Chapter 5: Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 2 145

WOW! eBook
www.wowebook.org

Client-side development using JavaScript 146
Preliminary email confirmation functionality 147
Email confirmation by our user 149
Using XMLHttpRequest 153
Optimizing your web applications and using bundling and minification 158
Bundling and minification in action 159
Working with WebSockets for real-time communication scenarios 163
WebSockets in action 164
Taking advantage of session and user cache management 168
In-memory session providers 169
Distributed session providers 173
Applying globalization and localization for multi-lingual user interfaces 174
Globalization and localization concepts 174
Using the view localizer 181
Localizing Data Annotations 184
Configuring your applications and services 188
Adding an email service 188
Configuring the email service 190
Implementing advanced dependency injection concepts 194
Method injection 194
Summary 201

Chapter 6: Introducing Razor Components and SignalR 202
Client-side development using C# Razor components 203
Working with SignalR 207
What is SignalR 207
SignalR with server-side Blazor or Razor components 208
Using logging and telemetry for monitoring and supervision purposes 209
Building once and running on multiple environments 221
Summary 227

Chapter 7: Creating ASP.NET Core MVC Applications 228
Understanding the Model View Controller pattern 229
Models 230
Views 231
Controllers 231
Unit tests 231
Integration tests 232
Creating dedicated layouts for multiple devices 232
The layout page in more detail 233
Optimizing for mobile devices 236
Understanding ASP.NET Core state management 241

WOW! eBook
www.wowebook.org

Client-state management options 242
Hidden fields 242
Cookies 242
Query string 243
Query string usage 243
Server-based state management options 244
Application state 244
Session state 244
Using view pages, partial views, View Components, and Tag Helpers 245
Using view pages 246
Using partial views 253
Using View Components 254
Using Tag Helpers 260
Dividing a web application into multiple areas 266
Applying advanced concepts such as view engines, unit tests, and integration tests 270
Using view engines 270
Providing better quality by creating unit tests and integration tests 277
Adding unit tests 282
Adding integration tests 286
Layering ASP.NET Core 3 applications 288
Determining the required layers 289
Deciding on the distribution for layers and components 290
Determining rules for interactions between layers 290
Identifying cross-cutting concerns 291
Summary 291

Chapter 8: Creating Web API Applications 293
Technical requirements 294
Applying web API concepts and best practices 294
Building RPC-style web APIs 296
Building REST-style web APIs 312
Building HATEOAS-style web APIs 321
Securing your web API 323
ASP.NET Core web API help pages with Swagger/OpenAPI 324
Summary 328

Section 3: Section 3: The ASP.NET Core 3 Supporting Ecosystem

Chapter 9: Accessing Data Using Entity Framework Core 3 330
Establishing a connection 333
Defining primary keys and foreign keys via Data Annotations 336
Using Entity Framework Core 3 migrations 340
Creating, reading, updating, and deleting data 344

WOW! eBook
www.wowebook.org

Understanding data relationships 346
Primary key 346
Foreign key 347
One-to-one relationships 348
One-to-many relationships 348
Many-to-many relationships 349
Working with queries 350
Querying for one item 350
Querying for all items 351
Querying for filtered items 351
Using transactions 352
Summary 353

Chapter 10: Securing ASP.NET Core 3 Applications 354
Implementing authentication 355
Adding basic user form authentication 369
Adding external provider authentication 377
Working with two-factor authentication 380
Two-factor authentication - step by step 381
Adding forgotten password and password reset mechanisms 390
Implementing authorization 399
Summary 408

Chapter 11: Securing ASP.NET Applications - Vulnerabilities 409
Cross-Site Scripting (XSS) 410
Preventing XSS 411
Cookie stealing 412
Preventing cookie stealing 412
Eavesdropping, message tampering, and message replay 413
Preventing eavesdropping and message replay 413
Open redirects/XSR 414
Open redirects example 414
Preventing open redirects 415
SQL injection 415
Preventing SQL injection 415
Protecting SQL connection strings 416
Using the Persist Security Info default value in connection strings 416
Using object-relational mappers (ORMs) 416
Cross-Site Request Forgery (XSRF/CSRF) 417
XSRF/CSRF example 417
Preventing XSRF/CSRF 418
Domain referrers 418
User-generated tokens 419
Limitations 420
JS/JSON hijacking 420
Preventing JSON hijacking 420

WOW! eBook
www.wowebook.org

Over-posting 420
Vulnerability example 421
Preventing over-posting 421
Clickjacking 422
Clickjacking example 422
Preventing clickjacking 422
Proper error reporting and stack trace 424
Error reporting vulnerability example 425
Preventing a screen of death 425
Summary 426

Chapter 12: Hosting ASP.NET Core 3 Applications 427
Hosting applications 428
Deploying applications in AWS 430
Deploying applications in AWS Elastic Beanstalk 433
Getting the application running on AWS 445
Deploying applications in Microsoft Azure 458
Deploying applications in Microsoft Azure App Service 462
Getting an Azure App Service instance running 462
Publishing your code on Azure 468
Continuous integration with Azure Repos 469
Connecting the database 472
Deployment through the Web Deploy tool 479
Deploying applications into Docker containers 482
Deploying applications into Docker containers 483
Publishing images to Docker Hub 491
Summary 494

Chapter 13: Managing ASP.NET Core 3 Applications 495
Logging in ASP.NET Core 3 applications 496
Logging in Microsoft Azure 497
Enabling Microsoft Azure App Service 498
Logging in AWS 505
Monitoring ASP.NET Core 3 applications 508
Monitoring on-premises and in Docker 509
Monitoring in Microsoft Azure 512
Monitoring in AWS 528
Summary 533

Other Books You May Enjoy 534

Index 537

WOW! eBook
www.wowebook.org

Preface

Every day, software developers, application architects, and IT project managers work on building applications as quickly as possible in order to be leaders in their respective markets: **time-to-market** (**TTM**) is of utmost importance. Unfortunately, the quality and performance of those applications are often not as expected, since they have not been fully tested, optimized, and secured.

During the past few years, ASP.NET has evolved into becoming one of the most consistent, stable, and feature-rich frameworks available on the market for web application development. It provides all expected characteristics you can think of concerning performance, stability, and security out of the box.

For some time now, the IT market has been changing. Compliance with different standards is now required and customers expect industrialized, high-performing, and scalable applications, while developers ask for frameworks that allow higher productivity and extensibility to adapt to specific business needs. Accordingly, this has led Microsoft to completely rethink its web technologies.

As a result, Microsoft has built ASP.NET Core, which gives developers the capacity to do the following:

- Create applications and compile them in a specific environment, but then run them in any environment (such as Linux, Windows, or macOS).
- Use third-party libraries with additional functionalities.
- Work with various tools, frameworks, and libraries.
- Adopt the most up-to-date best practices for frontend development.
- Develop flexible, responsive web applications.

ASP.NET Core 3, together with Microsoft Visual Studio 2019, includes several features to make your life as a web developer easier and more productive. For example, Visual Studio offers project templates that you can use to develop your web applications. Visual Studio also supports several development modes, including using Microsoft **Internet Information Services** (**IIS**) directly to test your web applications during development time and using a built-in web server to develop your web applications over FTP.

WOW! eBook
www.wowebook.org

With the debugger in Visual Studio, you can run through your application and step through the critical areas of your code to find problems. With the Visual Studio editor, you can effectively develop your own custom user interfaces.

And when you are ready to deploy your application, Visual Studio makes it easy to create a deployment package for deployment on Azure, Amazon Web Services, and Docker, or any other platform including Linux and macOS. These are but a few of the features built into the ASP.NET Core framework when paired with Visual Studio.

This book provides the latest best practices and ASP.NET Core guidance to get you up to speed quickly. Each section of this book presents specific ASP.NET Core 3 features in an easily readable format with detailed examples. The step-by-step instructions yield immediate working results. Most of the key features of ASP.NET Core are illustrated using succinct, easily understandable, and reusable examples. The examples are in-depth, in order to illustrate features without being overbearing.

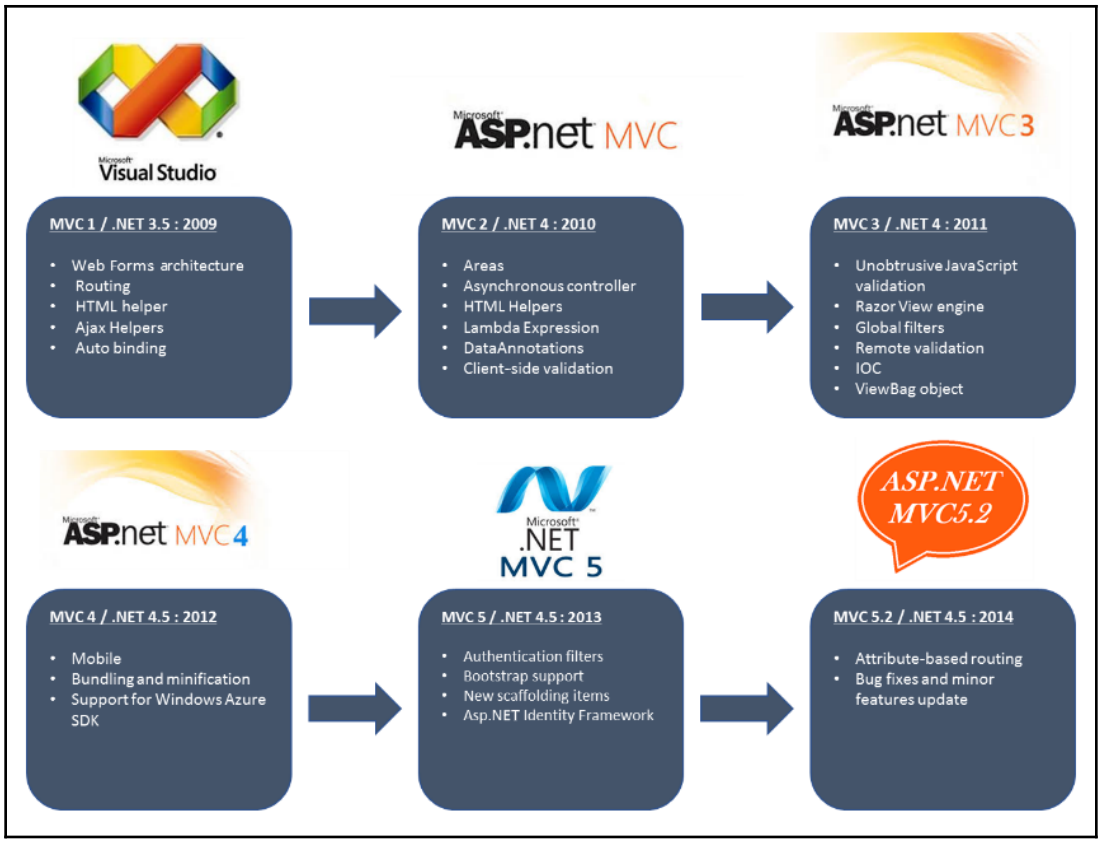
In addition to showing ASP.NET Core features by example, this book contains practical applications of each feature so that you can apply these techniques in the real world. After reading this book and applying the exercises, you will have a great head start into building efficient web applications that include modern features, such as MVC architectures, web APIs, custom view components, and tag helpers.

We hope this book will help you in your daily job as a developer and that reading it will give you as much joy as writing it has given us.

Once upon a time – NGWS and .NET Framework

The following is a little bit of history to explain how .NET Framework has evolved over the years and why you have to consider the .NET Core framework today:

WOW! eBook
www.wowebook.org



Microsoft started working on what we know now as .NET Framework in the late 1990s, and released a first beta version of .NET Framework 1.0 in late 2001.

Originally, the framework was named **NGWS** for **Next Generation Windows Services** (with an internal codename of Lightning/Project 42). In the beginning, developers could only use VB.NET as a programming language. More than 10 Framework versions later, a lot has been achieved. Today, you can choose between a large number of languages, frameworks, and technologies.

In the beginning, InterDev was the primary development environment to develop ASP pages, and you had to use a command-line VBC compiler tool to compile your code.

The first version of our beloved Visual Studio development environment was published in February 2002, bringing with it a common runtime environment for the Windows client and Windows server family (NT 4, Windows 98, Windows ME, Windows XP, and then Windows 2000).

WOW! eBook
www.wowebook.org

Around the same time, Microsoft provided a lighter framework, named Compact Framework, to execute Windows CE on Windows Mobile. The last version was published in January 2008 as version 3.5 RTM before it was replaced by newer mobile technologies.

The first .NET SDK was published in April 2003 as .NET Framework 1.1 and was included in Visual Studio 2003. It was the first version to be included in the Windows Server OS and shipped together with Windows 2003.

.NET Framework 2.0 was released in January 2006 during the time of Windows 98 and Windows Me. It provided a major upgrade to the **Common Language Runtime** (**CLR**). It was the first version to fully support 64-bit computing and fully integrate with Microsoft SQL Server. It also introduced a new Web Pages Framework, providing features such as skins, templates, master pages, and style sheets.

.NET Framework 3 (WinFX) was released in November 2006. It included a new set of managed code APIs. This version added several new technologies to build new types of applications, such as **Windows Presentation Foundation** (**WPF**), **Windows Communication Foundation** (**WCF**), **Windows Workflow Foundation** (**WWF**), and Windows CardSpace (later integrated into Windows Identity Foundation).

.NET Framework 3.5 extended the WinFX features one year later, in 2007. This version included key features such as LINQ, ADO.NET, ADO.NET Entity Framework, and ADO.NET Data Services. Furthermore, it shipped with two new assemblies that would later be the foundation of the MVC framework: `System.Web.Abstraction` and `System.Web.Routing`.

.NET Framework 4.0 was published in May 2009; it provided some major upgrades to the CLR and added a parallel extension to improve support parallel computing, dynamic dispatch, named parameters, and optional parameters, as well as code contracts and the `BigIntegerComplex` numeric format.

After the release of .NET Framework 4.0, Microsoft released a set of improvements to build microservices in the form of the Windows Server AppFabric framework. Essentially, it provided an in-memory distributed cache and an application server farm.

.NET Framework 4.5 was released in August 2012; it added a so-called Metro-style application (which later evolved into Universal Windows Platform applications), the Core features, and the **Microsoft Extension Framework** (**MEF**).

Concerning ASP.NET, this version was more compatible with HTML5, and jQuery, and provided bundling and minification for improved web page performance. It was also the first to support WebSockets and asynchronous HTTP requests and responses.

WOW! eBook
www.wowebook.org

.NET Framework 4.6.1 was released in November 2015; it required Windows 7 SP1 or later, and was an important version. Some of the new features and APIs included were support for SQL connectivity for AlwaysOn, Always Encrypted, and improved connection resiliency when using Azure SQL databases. It also added Azure SQL Database support for distributed transactions using the updated `System.Transactions` APIs and provided many other performance-, stability-, and reliability-related fixes in RyuJIT, GC, and WPF.

.NET Framework 4.6.2 was released in March 2016; it added support for paths longer than 260 characters, FIPS 186-3 DSA in X.509 certificates, and localization of data annotations, and the resources files were moved to the `App_LocalResources` folder. Additionally, the ASP.NET session provider and local cache manager were made compatible with the asynchronous framework.

.NET Framework 4.7 was released in April 2017; it was included in the Windows 10 Creators update. Some of the new features included enhanced cryptography with elliptic curve cryptography and improved **Transport Layer Security** (**TLS**) support, especially for version 1.2. It also introduced the object cache store, which enabled developers to provide custom providers easily by implementing the `ICacheStoreProvider` interface.

There was also a better integration between the application and the memory monitor and the famous memory limits reactions, which enables developers to observe the CLR when it truncates objects cached in memory and overrides the default behavior.

Then, Microsoft developed a completely new .NET Framework with open source multiplatform in mind from the beginning. It was introduced as ASP.NET 5 and later renamed ASP.NET Core Framework.

The first release, 1.0, was announced by Richard Lander (MSFT) in June 2016; the ASP.NET MVC and web API frameworks were merged into a single framework package that you could easily add to your projects via NuGet.

The second release, .NET Core Framework 1.1, was published in November 2017; it ran on more Linux distributions, its performance was improved, it was released with Kestrel, the deployment on Azure was simplified, and the productivity was improved. Entity Framework Core started to support SQL Server 2016.

The latest release of the .NET Core framework at the time of writing this book is 3, released in September 2019. A first preview version was released in late 2018 and subsequent multiple previews since the beginning of the year (2019).

Microsoft has vastly improved the .NET Core framework. The improvements and extensions are the results of the vision for .NET Core 3; it enables you to use more of your code in more places.

WOW! eBook
www.wowebook.org

It is worth noting that most of the regular libraries are available on GitHub. They can be forked and rebuilt by anyone who wants to extend or change any standard behaviors.

Who this book is for

This book is for developers who would like to build modern web applications using ASP.NET Core 3. No prior knowledge of ASP.NET or .NET Core is required. However, basic programming knowledge is assumed. Additionally, previous Visual Studio experience will be helpful but is not required, since detailed instructions will guide you through the samples of the book. This book can also help people who work in infrastructure engineering and operations to monitor and diagnose problems during the runtime of ASP.NET Core 3 web applications.

What this book covers

This book is organized into chapters that explain ASP.NET Core 3 features in an easy and understandable format with practical examples. Most of the key features of ASP.NET Core 3 are illustrated using succinct, efficient examples and step-by-step instructions to yield immediate working results.

You don't have to read the chapters in any order to find the book useful. Each chapter stands on its own, except for the first chapter, which details the fundamentals of ASP.NET Core—you might want to read it first if you've never ventured beyond desktop application development.

The following topics will be covered throughout the book.

`Chapter 1`, *What Is ASP.NET Core 3?*, describes the features and functionalities of ASP.NET Core 3, but also the technical restrictions, which should allow you to understand in which cases it could be a good fit for your own needs and what to expect.

`Chapter 2`, *Setting Up the Environment*, gives a detailed explanation of how to set up your development environment and how to create your first ASP.NET Core 3 application. You will learn how to either use Visual Studio 2019 or Visual Studio Code, how to install the runtime, and how to use NuGet to retrieve all necessary ASP.NET Core 3 dependencies.

`Chapter 3`, *Continuous Integration Pipeline in Azure DevOps*, demonstrates how to set up a complete Azure DevOps Continuous Integration Pipeline. You will learn how to fully automate building, testing, and deploying your applications using Azure DevOps in the cloud.

WOW! eBook
www.wowebook.org

`Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, explains the basic structure and concepts of ASP.NET Core 3 applications. It shows how everything works internally and what classes and methods can be used to override basic behavior. It also provides the theoretical background for all the other chapters.

`Chapter 5`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 2*, following up on the concepts covered in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, delves deeper into essential ASP.NET Core 3 concepts. You will learn about the components and features offered by ASP.NET Core to build responsive web applications.

`Chapter 6`, *Introducing Razor Components and SignalR*, gives an introduction to Blazor, a new offering by Microsoft to cater for frontend development using C# as a language. It prepares you with the basics that you need to be aware of what is being offered in working with server-side Blazor.

`Chapter 7`, *Creating ASP.NET Core MVC Applications*, provides all the concepts and everything necessary to create your first ASP.NET Core 3 MVC application. You will learn the specifics of MVC applications and how to implement them efficiently. Additionally, you will see how unit tests and integration tests will help you build better applications with fewer bugs, resulting in lower maintenance costs.

`Chapter 8`, *Creating Web API Applications*, covers the web API framework and provides everything essential to create your first ASP.NET Core 3 web API. You will see different web API styles, such as RPC, REST, and HATEOAS, and learn when to use them and how to implement them in an effective way.

`Chapter 9`, *Accessing Data Using Entity Framework Core 3*, shows how to access databases using Entity Framework Core 3, while using all the advanced features (code first, the Fluent API, data migrations, in-memory databases, and more) it offers.

`Chapter 10`, *Securing ASP.NET Core 3 Applications*, explains how to use the built-in ASP.NET Core 3 features for user authentication and how to extend them by adding external providers. If you need to secure your applications, then this chapter is where you want to go.

`Chapter 11`, *Securing ASP.NET Applications - Vulnerabilities*, gives us an indication of what we need to be aware of when building our applications, in terms of areas that can be exploited, and therefore need more attention.

WOW! eBook
www.wowebook.org

`Chapter 12`, *Hosting ASP.NET Core 3 Applications*, is about the various options you have when it comes to hosting and deploying your ASP.NET Core 3 web applications on-premises and in the cloud. You will learn how to choose the appropriate solutions for a given use case, which will allow you to make better decisions for your own applications.

`Chapter 13`, *Managing ASP.NET Core 3 Applications*, is finally going to be a chapter on how to manage and supervise your production-ready applications after deployment. It will greatly aid you in diagnosing problems for your ASP.NET Core 3 web applications during runtime and reduce the time to understand and fix bugs.

To get the most out of this book

You will either need Visual Studio 2019 Community Edition or Visual Studio Code, which are both free of charge for testing and learning purposes, to be able to follow the code examples found within this book. You could also use any other text editor of your choice and then use the `dotnet` command-line tool, but it is advised to use one of the development environments mentioned earlier for better productivity.

Later in the book, we will work with databases, so you will also need a version of SQL Server (any version in any edition will work). We advise using SQL Server 2019 Express Edition, which is also free of charge for testing purposes.

There might be other tools or frameworks that will be introduced during the following chapters. We will explain how to retrieve them when they are used.

If you need to develop for Linux, then Visual Studio Code and SQL Server 2016 or 2019 are your primary choices, since they are the only ones running on Linux.

Additionally, you will need an Azure subscription and Amazon Web Services subscription for some of the examples shown within the book. There are multiple chapters dedicated to showing you how to take advantage of the cloud.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

WOW! eBook
www.wowebook.org

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for macOS
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at the following repository: `https://github.com/PacktPublishing/Learn-ASP.NET-Core-3-Second-Edition`.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789610130_ColorImages.pdf`.

Code in Action

Please visit the following link to see the Code in Action videos: `http://bit.ly/39ecHAf`.

Conventions used

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

WOW! eBook
www.wowebook.org

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. "Start Visual Studio 2019, open the Tic-Tac-Toe ASP.NET Core 3 project you have created, create three new folders called `Controllers`, `Services`, and `Views`, and then create a subfolder called `Shared` in the `Views` folder."

A block of code is set as follows:

```
[HttpGet]
public IActionResult EmailConfirmation (string email)
{
  ViewBag.Email = email;
  return View();
}
```

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

```
public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public StudentDetails StudentDetails { get; set; }
    public ICollection<StudentSubject> StudentSubjects { get; set; }
        // Added after constructed table
}
```

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

```
sudo apt-get install code
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Open Visual Studio 2019, go to the **Team Explorer** tab, and click on the **Branches** button."

Warnings or important notes appear like this.

Tips and tricks appear like this.

WOW! eBook
www.wowebook.org

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

WOW! eBook
www.wowebook.org

1
Section 1: Introduction and Environment Setup

This section will gracefully introduce you to ASP.NET Core 3 and its features, as well as when it makes sense to use the web framework. By the end of this section, you will understand the capabilities of ASP.NET Core 3 and have set up a development environment where the demo application that we'll be using throughout this book will be developed.

This section comprise the following chapters:

- `Chapter 1`, *What Is ASP.NET Core 3?*
- `Chapter 2`, *Setting Up the Environment*
- `Chapter 3`, *Continuous Integration Pipeline in Azure DevOps*

WOW! eBook
www.wowebook.org

1
What Is ASP.NET Core 3?

The world's very first form of a web server came into being around the year 1990. It was called **CERN httpd** and was developed by Tim Berners-Lee, a name quite synonymous with the origins of the **World Wide Web** (**WWW**).

A web server in its rudimentary form was supposed to handle requests that only expected the contents of a file as a response, but, with time, there have been additions in expectations, with a change in the initial need for static files only as opposed to today's dynamic web applications, which are more demanding.

Nowadays, there are blurred lines on what responsibility a web server has, as opposed to a web application, and these lines become clearer as we learn more about ASP.NET Core 3 as a web application framework. It is worth noting that this and other frameworks not originating from Microsoft, such as Ruby on Rails, act as a buffer between you as a developer and the web server.

ASP.NET Core 3 includes several benefits over other previous frameworks and the advantages are elaborated on in subsequent sections when we take a closer look at its features and what is new specifically to this version.

In this chapter, we will cover the following topics, with an obvious bias toward ASP.NET Core 3:

- The history of ASP.NET
- ASP.NET Core 3 features
- What is new specifically to ASP.NET Core 3?
- Cross-platform support
- Microservice architecture
- Performance and scalability
- Technology restrictions
- When to choose ASP.NET Core 3

WOW! eBook
www.wowebook.org

The history of ASP.NET

It all began with **Active Server Pages** in the mid-nineties, with Microsoft trying to keep up to date with the buzz of serving dynamic content over the web at that time, and that obviously influenced the name **Active Server Pages**, conventionally known today as ASP.

As with any worthwhile technology, ASP.NET has been evolving over time with one of the major shifts being the introduction of **ASP.NET Web Forms** around the year 2002, which was influenced heavily by the success of another one of Microsoft's application frameworks meant for the desktop environment, called **Windows Forms**, or more commonly known as **WinForms**.

With the ease of creating HTML forms and controls in WinForms came a lot of baggage of unnecessary HTML and JavaScript that slowed down page loading, along with other factors such as view state and page life cycle that contributed further to slowing down business applications. This led to the introduction of a series of ASP.NET MVC versions that tried to solve some of the problems that ASP.NET Web Forms had.

ASP.NET MVC also helped to cater to one of the major tenets of good programming practices in preferring **separation of concerns** (**SoC**) over the tight coupling that was evident in ASP.NET Web Forms with its **code-behind** files. This had in itself introduced ripple benefits in allowing for test-driven development and improving testability in general.

Another major shift happened in 2016 with the release of **ASP.NET Core** in its first version, 1.0, which has continued to evolve up to version 3 at the time of writing this book (2019). In this shift, Microsoft almost completely rewrote ASP.NET, mainly removing its dependency on the `System.Web` **namespace**, which necessitated a reliance on **Internet Information Services** (**IIS**). Since IIS is compatible only with the Windows operating system, independence from it allowed ASP.NET Core to be truly cross-platform.

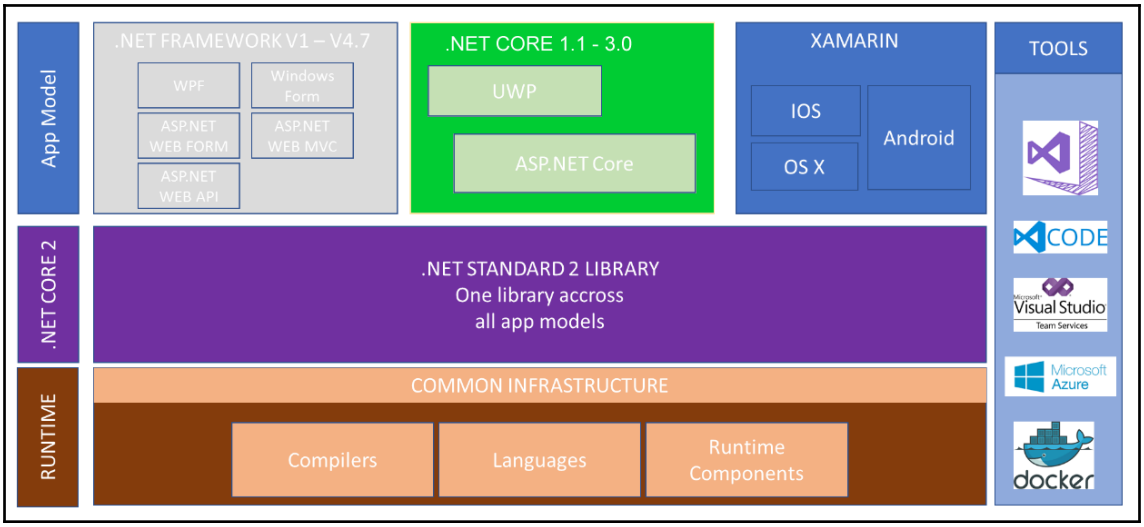
It must be mentioned that Microsoft embraced **open source** visibly from around the year 2014 with a change in business dynamics and while one of the biggest selling points of ASP.NET Core is that it is open source, we need to be aware that even the previous versions of ASP.NET, including MVC and the web API, were also eventually released as open source, and anyone can contribute to their continued development:

ASP.NET Version	Year From
Active Server Pages	1996
ASP.NET Web Forms	2002
ASP.NET MVC 3,4,5,6	2008
ASP.NET Core 1.0, 1.1, 2.0, 2.1, 2.2, 3.0	2016

WOW! eBook
www.wowebook.org

Before version 3, ASP.NET Core applications ran on the .NET Core framework as well as on full .NET Framework, but a decision was made by Microsoft that, starting from version 3, ASP.NET Core would run only on .NET Core, to make better use of new developments, without being tied down to catering for old functionality.

In the following diagram, you can see how the different .NET Framework versions and components work together:



This book is about ASP.NET Core, and more specifically, its latest version, 3 (at the time of writing). Therefore, the brief mention earlier of the previous versions suffices to just give us context, but from now on, we will focus a bit more on ASP.NET Core.

Our focus in this book remains ASP.NET Core 3, which is different from .NET Core 3; the former being an application framework and the latter being a runtime. An ASP.NET Core application is traditionally able to run on .NET Core as well as other .NET Framework versions, and that underlines the fact that they are different.

It is easy to see why some people confuse the two because an ASP.NET Core application can also be a .NET Core application, the same way it can be a .NET Framework 4.8 application.

It is quite important to note when making decisions about what framework to use for developing new applications that Microsoft has plans for future releases of ASP.NET Core to only run on .NET Core and not other .NET Framework versions.

WOW! eBook
www.wowebook.org

Having looked at a brief history of ASP.NET Core 3, let's have a look at the features that define the application framework in the next section.

ASP.NET Core 3 features

The `Microsoft.AspNet.Core.All` package in ASP.NET Core 2.0 contains all features in a single library. It includes authentication, **Model-View-Controller** (**MVC**), Razor, monitoring, Kestrel support, and much more.

Referencing `Microsoft.ASP.Net.Core.All` as a package has been discouraged since ASP.NET Core version 2.1, and this applies to the current version, 3.

We can still use the namespace by making use of patches, but the preferred replacement is the `Microsoft.AspNetCore.App` shared framework, details of which are explained in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, when we expound the basic concepts of ASP.NET Core 3.

In an effort to make ASP.NET Core as lightweight as possible, and perhaps for better control, Microsoft decided only to let assemblies developed and maintainable in-house to be in the shared framework, and excluded third-party assemblies that were available with the `Microsoft.AspNet.Core.All` namespace.

Notable casualties that are not fully owned and therefore not fully controlled by Microsoft that were removed from the framework include Json.NET. We are, however, still able to use them by adding their references.

ASP.NET Core 3 also allows us to create applications that follow the MVC architectural style, with a ready-made template that is available for use and we have dedicated a full chapter to this topic later in this book.

Furthermore, we can build HTTP-based web services as well as RESTful services. A new addition to the capability in implementing microservices, the **gRPC** template, that comes with ASP.NET Core 3 is introduced in the next section on what is new specifically to ASP.NET Core 3.

ASP.NET Core 3 fully supports Razor, which contains an efficient language for creating our views and Tag Helpers, which allow logic to be written from the server side to generate HTML that can be used in Razor views.

WOW! eBook
www.wowebook.org

In terms of client-side development, ASP.NET Core 3 integrates and works hand in hand with several frameworks external to Microsoft including Angular, React, and React-Redux, although it must be noted that these will become less and less prominent, with an obvious attempt by Microsoft to handle similar functionality with in-house Razor components, otherwise known as **Blazor**.

Additionally, ASP.NET Core provides the following fundamental improvements:

- ASP.NET MVC and the web API have been combined into a single framework.
- The environment-based configuration system is ready for cloud hosting.
- Dependency injection functionalities come by default.
- You can host the same application in IIS, Docker, the cloud, and even in your own processes, or you can self-host.
- There's new tooling that simplifies modern web development.
- There's a simplified `csproj` file, making it easier to work with development environments other than Visual Studio (on Linux and macOS, for example).
- `Startup.cs` has been simplified by moving logging and configuration into the host builder initialization.
- ASP.NET Core apps can now be developed on Visual Studio for Mac.

The features we have seen could apply also to other versions of ASP.NET Core prior to 3, but others apply only to version 3 and maybe higher, in the future. We will look at them in the next section.

What is new specifically to ASP.NET Core 3?

ASP.NET Core 3 exists in an ecosystem where everything else is changing as well, including the .NET Core runtime, which is currently at version 3 as well, and the C# language itself, which is at version 8. With all of these changes, ASP.NET Core has been adapting to the ecosystem changes as well, not only to Microsoft-related changes but also taking into consideration the developer community in general, as evidenced by, for example, changing the Angular template to Angular 7.

The ASP.NET Core 3 shared framework has been made significantly more lightweight, being decoupled from other non-core components such as Entity Framework Core, Roslyn code analysis, and Json.NET.

Changes such as these have inevitably effected other changes in a ripple effect, for example, in the forced removal of **runtime compilation**, which is obviously made possible by Roslyn, and as such, ASP.NET Core 3 is significantly more lightweight than its predecessors.

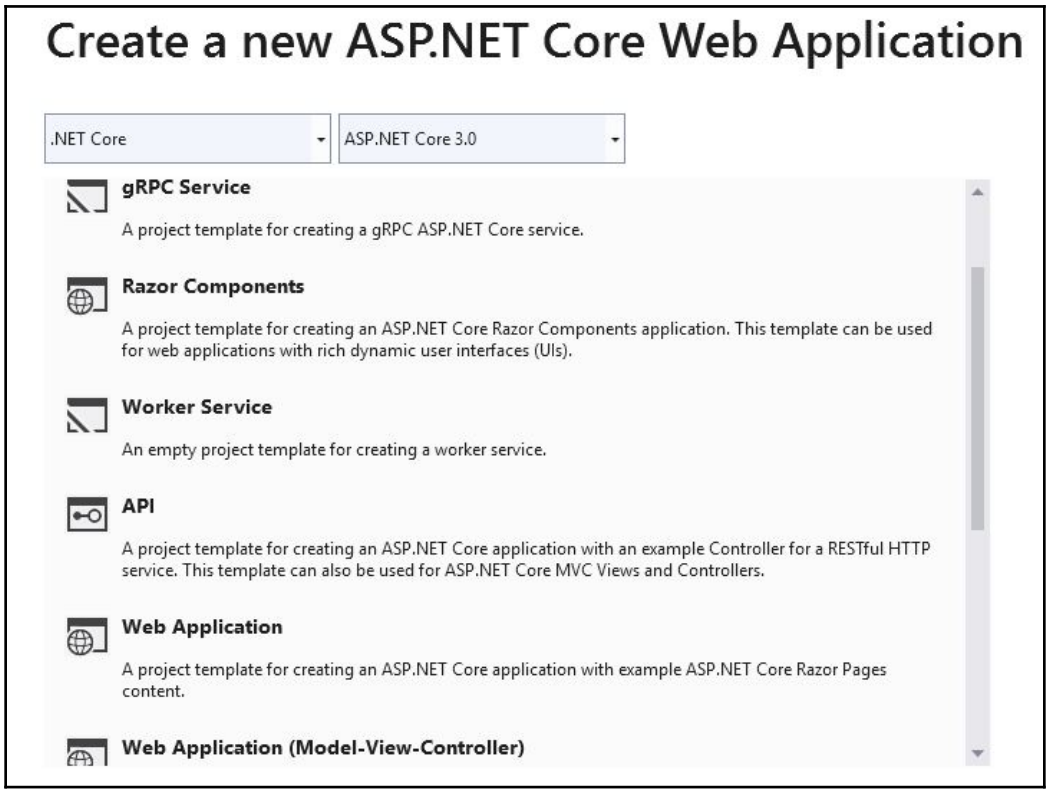
WOW! eBook
www.wowebook.org

ASP.NET Core evolved from being referenced as packages in 1.0 into being a shared framework in 2.1. However, in 3, we no longer reference `Microsoft.AspNetCore.App` through the `<PackageReference>` element, which is naturally replaced as `<FrameworkReference>`.

If your project references the `Microsoft.NET.Sdk.Web` SDK, then it automatically has access to the shared framework. Commonly referenced APIs such as MVC, Razor, and Kestrel, among others, are no longer referenced as **NuGet** packages but are still available to us as developers through the same `<FrameworkReference>` element to `Microsoft.AspNetCore.App`.

ASP.NET Core 3 has attempted to improve integration with **OpenAPI** and has introduced a system for generating API clients that integrate easily with **NSwag** and other code generators.

The following screenshot shows some of the new templates introduced from ASP.NET Core version 3:



WOW! eBook
www.wowebook.org

One of the biggest introductions to ASP.NET Core 3 is the **C# Razor components**, which has, to date, been known as Blazor. It was being developed separately as an independent experimental framework, and with it comes a new `.razor` extension that helps the compiler to identify a file with Razor components.

Normally, JavaScript code is what most browsers have been able to understand and execute, but what Razor components bring to the table is the ability to be able to run C# in the browser. We will talk a bit more about Razor components in `Chapter 6`, *Introducing Razor Components and SignalR*.

ASP.NET Core 3 comes with a **Worker Service** template by default. If you're coming from a desktop development background, you will be familiar with Windows Services, and similarly daemons for those with Linux experience, and as an answer for the web environment, ASP.NET Core 3 has introduced a template for us to be able to develop worker services to cater for long-running services.

Another exciting new feature added with ASP.NET Core 3 is the **gRPC Service template**, which is going to be popular with developers who use microservices often. **gRPC** originated from Google and uses a bit more lightweight protocol buffer serialization compared to the common XML/JSON serialization in service-to-service communication over HTTP/2. A demonstration of this will be included in `Chapter 8`, *Creating Web API Applications*.

There have been significant improvements to the routing model as previous users of ASP.NET MVC will know, mainly with how it operates with middleware, aptly referred to as **endpoint routing**; it was introduced in 2.2 but specifically introduced in 3 is SignalR and Razor component integration with **endpoint routing**. More about this will be covered in `Chapter 6`, *Introducing Razor Components and SignalR.*

Now that we have seen many important features associated with ASP.NET Core 3, there is one feature that deserves a special mention and specific coverage, because of its significance. In today's diverse technological platforms, it is hugely important to support different platforms, and therefore we will look at how ASP.NET Core 3 is geared up for cross-platform support in the next section.

Cross-platform support

As explained before, the ASP.NET Core 3 framework has been built, from the beginning, with cross-platform support in mind. It supports a wide variety of operating systems and technologies such as Windows, Linux, macOS, Docker, Azure, and others.

WOW! eBook
www.wowebook.org

ASP.NET Core 3 currently supports the following Linux distributions:

- Ubuntu 14, 16
- Linux Mint 17, 18
- Debian 8
- Fedora
- CentOS 7.1 and Oracle 7.1
- SUSE Enterprise Server 64 bits
- openSUSE 64 bits

Concerning macOS, it currently only supports the following (other versions might be added later):

- macOS 10.11
- macOS 10.12

For application development, you may develop on Windows using Visual Studio or Visual Studio Code and then deploy your ASP.NET Core 3 application to your target system.

Note that the target system can use a completely different underlying operating system. For instance, you can develop and test on Windows and then deploy your applications to a Linux server for performance, stability, or cost reduction reasons.

If you choose so, you can of course directly develop on Linux and macOS using several system-specific source code editors. On Linux, you could use Visual Studio Code, Vim/Vi, Sublime, or Emacs, for example. On macOS, you could use Visual Studio for Mac, Visual Studio Code, or any other macOS-specific text editor.

The Visual Studio 2019 or Visual Studio Code developer environments would be the preferred choice, though, since they provide everything necessary to be highly productive and to be able to debug and understand your code as well as navigate within it easily. That is why we are going to use those IDEs throughout the rest of this book.

After building your application, you can use several web servers to run it. Here are some examples:

- Apache
- IIS
- Kestrel self-host
- NGINX

WOW! eBook
www.wowebook.org

Cross-platform is a huge factor, and we have seen how ASP.NET Core 3 caters to it, but there's another buzzword in the software engineering fraternity called microservices. Let's have a look at it with respect to ASP.NET Core 3 in the next section.

Microservice architecture

Microservice architecture, most commonly referred to as just *microservices*, is a currently common way of designing and building software applications in a modular way with the single responsibility principle in mind. It stresses having service modules that are not tightly coupled with other services when implementing business solutions that are service-oriented. Microservices can be used to build e-commerce systems, business applications, and IoT. You will find them quite a popular implementation especially when working with distributed applications.

ASP.NET Core 3 is the best candidate when you want to embrace this system architecture. The ASP.NET Core 3 framework is lightweight and its API surface can be minimized to the scope of a specific microservice. A microservice architecture also allows you to mix technologies across service boundaries, enabling for a gradual transition to ASP.NET Core.

Notice that microservices built with ASP.NET Core 3 can work together with services using other technologies such as the full classic .NET Framework, Java, Ruby, and even other more legacy technologies. This is a big advantage when you need to progressively transform monolithic applications into more (micro)service-oriented applications.

You are not bound to the specific underlying infrastructure; instead, you have a wide range of choices since ASP.NET Core 3 supports nearly all of the technologies that you can think of today. Additionally, you can modify the infrastructure when needed so there is no technological lock-in for applications that have been developed based on it.

Your primary choice for orchestrating and managing microservices written in C# efficiently and at a high scale, on-premises, and in the cloud should be Microsoft Service Fabric, also known as Azure Service Fabric. It was conceived exactly for that and has been used by Microsoft for various Azure services (such as SQL Database) for many years already.

A microservices Docker container approach might also fit your needs, and we are going to explain its use cases in the next section. To sum it up, ASP.NET Core 3 is the ideal choice for implementing and hosting your microservices in any kind of technical environment.

WOW! eBook
www.wowebook.org

Working with containers

Containers are popular at the moment as they provide an efficient, lightweight, and self-contained approach for packaging applications with their dependencies while re-using the underlying operating system files and resources.

They are a perfect fit for microservice architectures, but can also be used for any other application archetypes. They work exceptionally well together with ASP.NET Core 3 applications since both have been conceived with modularity, performance, scalability, lightweight nature, and efficiency in mind.

We must note that there are currently different containers available for use by the developer community such as CoreOS rkt, Apache Mesos Containerizers, and **LXC** (short for **Linux Containers**), but the most popular by far are Docker containers.

Note that Docker container images including ASP.NET Core 3 applications are much smaller than images with classic ASP.NET applications, meaning that they are faster to deploy and to start up.

Both Docker containers and the ASP.NET Core 3 framework provide full cross-platform support (Windows, Linux, and macOS). Furthermore, you can host your containers on-premises and in the cloud. You can use Azure, for example, either via **Infrastructure-as-a-Service** (**IaaS**) deployments or via **Azure Container Service**, which is being deprecated in favor of **Azure Kubernetes Service**, which additionally allows for mixing and matching different operating systems and technologies.

Microservices architecture, cross-platform support, and other features might make ASP.NET Core 3 a great framework to use, but how good is it if it has such great features without a matching great performance? How does ASP.NET Core 3 fare in terms of being able to handle applications that need to grow? We will look at both performance and scalability for ASP.NET Core 3 in the next section.

Performance and scalability

If you need the best possible performance and support for high-scalability scenarios, then you absolutely need to use ASP.NET Core 3 and the underlying .NET Core Framework currently in version .NET Core 3.

ASP.NET Core 3 has been built from the ground up for high-performance and high-scalability scenarios. It really shines in these areas and it can be considered as the best choice.

WOW! eBook
www.wowebook.org

It is many times faster than classic ASP.NET and can be thought of as the fastest web application runtime in the .NET world currently available!

If we are to go by the tests done by TechEmpower, which measure the performance of different web frameworks, found here: `https://www.techempower.com/benchmarks`, you will note that ASP.NET Core definitely comes out top compared to its .NET peers, and certainly does quite well too against its competitor frameworks by other providers:

Best plaintext responses per second, Test environment (331 tests)

Rnk	Framework	Best performance (higher is better)		Cls	Lng
3	aspcore	7,000,118	100.0%	Plt	C#
23	netty	4,637,485	66.2%	Plt	Jav
25	fasthttp	4,592,321	65.6%	Plt	Go
27	undertow	4,470,714	63.8%	Plt	Jav
31	nginx	3,940,906	56.3%	Plt	C
46	servlet	2,387,701	34.1%	Plt	Jav
82	go-prefork	951,413	13.6%	Plt	Go
89	nodejs	867,972	12.4%	Plt	JS

You can run benchmarks for ASP.NET Core using the details found on Microsoft's ASP.NET Core benchmarks project here: `https://github.com/aspnet/benchmarks`.

Furthermore, it provides the best solution for microservices architectures, where performance and scalability are extremely important. No other technology is as efficient while consuming such low system resources, which also leads to reduced infrastructure and cloud hosting costs.

We have so far seen how great using ASP.NET Core 3 as a platform can be, with all of the features mentioned earlier, but unfortunately, other technologies are not supported by the platform and its runtime. We look at them in the next section.

Technology restrictions

Please look carefully at the technologies shown in this section. If you use a technology or framework within your current application that is listed here and that is not (yet) supported, then you might find it difficult or even impossible to migrate to ASP.NET Core 3.

WOW! eBook
www.wowebook.org

Not all current .NET Framework technologies are available in ASP.NET Core 3 and some might never be ported over since they do not comply with the new .NET Core-specific paradigms and patterns.

Common technologies not directly found in ASP.NET Core and .NET Core

The following list shows the most common technologies not directly found in ASP.NET Core and .NET Core, though some can be used via multi-targeting features:

- **ASP.NET Web Forms applications**: The legacy Web Forms technology is only available using the full classic .NET Framework; you cannot use ASP.NET Core and .NET Core for these types of applications.
- **ASP.NET Web Pages applications**: They are not included in ASP.NET Core 3 as such, but it is possible to use the Razor web pages engine to provide the same functionalities.
- **WCF Services**: ASP.NET Core 3 contains a WCF client for accessing WCF services, but creating WCF services is not supported.

Not all of the templates available for ASP.NET Core 3 support all of the major .NET languages; for example, the only template available for VB.NET is GtkSharp, with F# having a few more templates, including the ASP.NET Core web API and F# TypeProvider templates. A more comprehensive list of what templates are available for what language can be found at this link: `https://github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new`.

When to choose ASP.NET Core 3

ASP.NET Core 3 and the underlying .NET Core Framework runtime indeed provide some major enhancements and performance improvements, but there are still some specific scenarios where those new application patterns do not apply and where the full .NET Framework will be the best and sometimes even the only choice.

Migrating your whole existing applications to ASP.NET Core right from the start might be difficult or even impossible to do. You should think about how to transform your applications progressively to lower the risk of failure or over complication and give yourself time to really understand the new patterns and paradigms.

WOW! eBook
www.wowebook.org

You could start for instance by only using ASP.NET Core 3 for all new developments, then see how to migrate your legacy code later and sometimes even leave it be since there will be no real benefits for migrating it over. If you are really interested in the migration topic, please consider the *Appendix* since we have a full chapter dedicated to this important topic.

ASP.NET Core and .NET Core Framework get more and more framework and client library support each day. Microsoft, tool and framework vendors, and the different developer communities work hard to provide a large set of functionalities for allowing feature-rich and high-performing web applications. Everybody wants to work on this promising technology that could sustainably shape the future.

The possibility to use .NET Core and .NET Framework libraries together at the same time when using .NET Standard 2.0 extends the possibilities even more and gives developers a temporary solution until every important feature and every major framework will be available in .NET Core.

To recap what has been discussed in this section, you should use ASP.NET Core 3 for your server applications if the following is true:

- You have cross-platform needs.
- You are specifically targeting microservices.
- You want to use Docker containers.
- You need high-performance and highly scalable applications.
- The presented technical restrictions do not apply to your application requirements.

Summary

In this chapter, you learned about the ASP.NET Core 3 framework and its features. You have seen that it includes everything necessary to work efficiently in a cross-platform environment while using microservices architectures and container technologies such as Docker.

Furthermore, you learned that it provides very good performance and exceptional scalability for your web applications.

At the end of this chapter, we talked about technical restrictions and when it is advisable to use the ASP.NET Core 3 framework.

In the next chapter, we will talk about how to set up your development environment including either Visual Studio 2019 or Visual Studio Code as an IDE.

WOW! eBook
www.wowebook.org

2
Setting Up the Environment

You have decided to learn about ASP.NET Core 3, the most advanced and efficient cross-platform web application framework on the market today. A very good choice! You are surely eager to start programming right away, but before we can begin, we must set up the required technical prerequisites and tools.

In this chapter, we are going to introduce Visual Studio 2019 Community Edition and Visual Studio Code, and then install either one of them as a development environment. Then, we are going to build a simple sample application based on the ASP.NET Core 3 Framework.

After going through the content in this chapter, you will be able to install different kinds of ASP.NET Core 3 development environment on the Windows operating system, macOS, and Linux. You will also learn about the basic debugging skills you'll need to troubleshoot most ASP.NET Core-based applications.

To sum up, in this chapter, we will cover the following topics:

- Visual Studio 2019 as a development environment
- How to install Visual Studio 2019 Community Edition
- Creating your first ASP.NET Core 3 application in Visual Studio and via the command line
- Basic debugging with Visual Studio 2019 Community Edition
- Visual Studio Code as a development environment
- How to install Visual Studio Code on Linux
- Creating your first ASP.NET Core 3 application in Visual Studio Code
- Creating your first ASP.NET Core 3 application in Linux
- Introduction to C# Interactive and LINQPad as tools

WOW! eBook
www.wowebook.org

Visual Studio 2019 as a development environment

As a developer, you need an environment for your daily development tasks, and Microsoft Visual Studio 2019 is just that.

There are many other IDEs available for developers across different programming languages, with some notable ones being NetBeans, PyCharm, IntelliJ IDEA, Eclipse, Code::Blocks, and XCode, among others. While you can use many of these to program using C#, which is the base language that we'll make use of in this book, it must be noted that the aforementioned IDEs are more suited for other languages, such as Python and Java.

Other IDEs that are more suitable for the C# programming languages include Visual Studio Code, MonoDevelop, SharpDevelop (#develop), JetBrains Rider, CodeMaid, and .NET Fiddle.

This book will make use of the two most commonly used IDEs for developers on the Microsoft tech stack: the Visual Studio series and Visual Studio Code.

Visual Studio 2019 provides a very efficient and productive **Integrated Development Environment** (**IDE**) for creating new software projects and developing, debugging, and testing them. It will help you build high-quality applications in a very quick and intuitive way. Many of its features have been built around common developmental tasks and how to streamline and optimize them within a single tool. By using this tool, you can create web applications, web services, desktop applications, mobile applications, and many other types of application that are not covered in this book.

Additionally, you can use a wide range of programming languages such as C#, Visual Basic, F#, JavaScript, and even Java or other languages that are not maintained by Microsoft.

There are different editions of Visual Studio 2019, each with its own unique features and licenses. The Visual Studio 2019 Community Edition, for instance, is free of charge but has fewer features than the Professional and Enterprise Editions, which we will explain later. The intended usage of the community version is for private use and learning purposes.

The Visual Studio 2019 Professional and Enterprise Editions contain far more features, including the necessary licenses to build and run applications in production environments.

The Visual Studio 2019 Professional Edition contains a subset of all the features that are offered in the Enterprise Edition. It is usually sufficient to start with this edition and then upgrade to the Enterprise Edition if necessary.

WOW! eBook
www.wowebook.org

Visual Studio 2019 Enterprise Edition contains a lot of additional features that we can use to improve developer productivity even more, such as time travel debugging, live dependency validation, live testing, architecture diagrams, architecture validation, code cloning, and many others. If you need these features, then you need to use this edition.

A full comparison of what features are available for each edition can be found at the following link: `https://visualstudio.microsoft.com/vs/compare/`.

Note that multiple versions of Visual Studio (2013, 2015, 2017, 2019, and more) can be installed side by side on a developer machine that has earlier versions of the Visual Studio IDE installed.

Traditionally, Visual Studio was released only for Windows, but a macOS version has existed since 2016 called Visual Studio for macOS. You can use it to develop your .NET applications on this operating system. Visual Studio for macOS supports developing .NET Core, ASP.NET Core, Mono library (including .NET Standard), and Xamarin apps. You can build the same kind of ASP.NET Core apps that you can in Visual Studio but the tooling is not as rich yet.

The Visual Studio 2019 Community Edition is exactly what we need for trying out and understanding the examples that will be illustrated in this book. This is exactly why we'll be installing it in the next section.

How to install Visual Studio 2019 Community Edition

Visual Studio 2019 Community Edition can be installed like any other Windows application.

Note that you need administrator rights during the installation process. These rights will not be required when developing with Visual Studio later.

WOW! eBook
www.wowebook.org

To install the Visual Studio 2019 Community Edition, you can choose between the following three different Visual Studio 2019 installation modes:

- The **express installation** installs all of the components that are considered default components by Microsoft in an easy and quick way. These components are found on the default **Workloads** tab and are conveniently grouped into Windows, web and cloud, mobile and gaming, and other toolsets, which are all available for you to install by just selecting the respective checkboxes. If you need specific Visual Studio features that aren't in this list, then you need to use the custom installation.
- The **custom installation** option gives you complete choice over every Visual Studio 2019 feature you can install. You may, for instance, install complementary features such as Visual C++, F#, SQL Server Data Tools, the mobile platform, and several other SDKs, as well as specific language packs through the **Individual components**, **Language packs**, and **Installation locations** tabs. Install groups in the VS Installer are called **workloads**.
- When using the **offline installation**, you can install Visual Studio 2019 without having a network connection. This is very handy when you cannot connect to the internet and nonetheless want to prepare a developer machine. In this case, you have to prepare external support, such as a mobile hard disk or a USB key, and put the Visual Studio 2019 installer files on it beforehand.

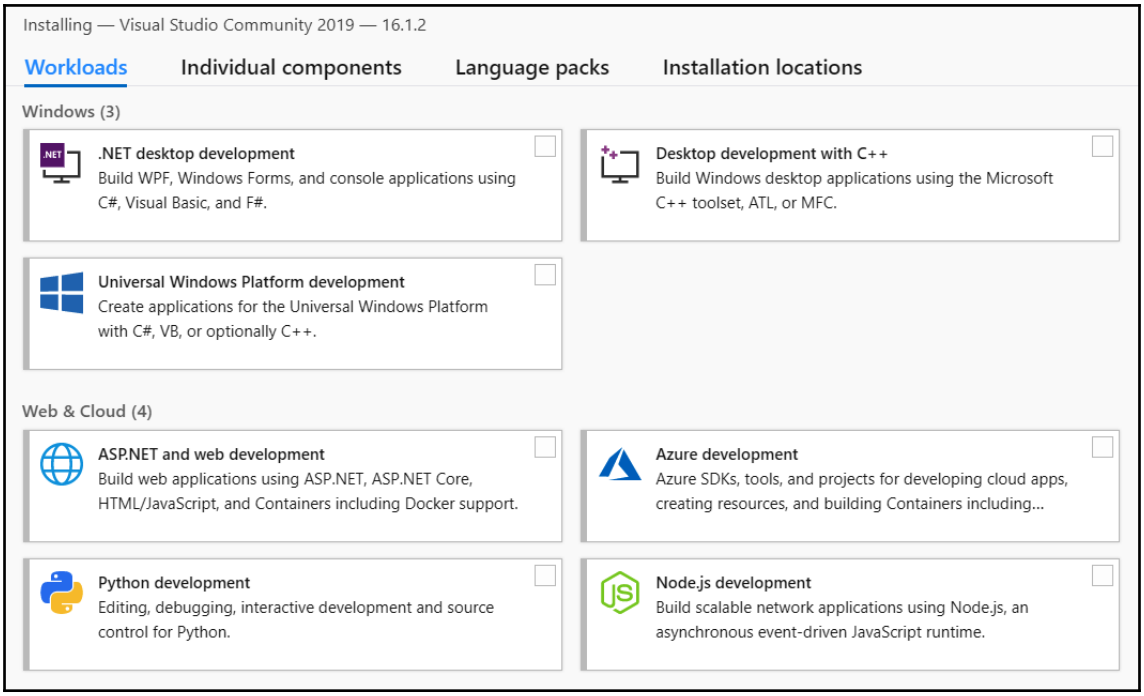
One way to prepare such external support is to download the necessary Visual Studio installer (Community, Professional, or Enterprise Edition) from the Visual Studio website, `https://www.visualstudio.com/downloads/`, and extract its contents into a folder. Then, you can retrieve the various install packages by executing the `<executable name> --layout` command in a command-line window. After some time, everything will be downloaded and you'll have external support that can be used for offline installations.

Note that you can use the same procedure to download all of the installation files to your central network storage and then create a shared folder so that you can install Visual Studio 2019 from within your own network to optimize installation times and lower network bandwidth needs.

WOW! eBook
www.wowebook.org

Now, let's learn how to install Visual Studio 2019 Community Edition manually using the setup program downloaded from the Microsoft Visual Studio website we mentioned previously:

1. Start the Visual Studio 2019 Community Edition setup program. You will see a list of various installable workloads. By default, you will see **Windows**, **Web and Cloud**, **Mobile and Gaming**, and **Other Toolsets**:



2. Choose your desired components – they will be installed in the following steps. If that is all you need, then you don't need to do anything else. As we explained previously, this is the express installation process.

WOW! eBook
www.wowebook.org

3. If you need to customize installed components or add or remove individual components, then you have to click on **Individual components**. From here, you will be doing a custom installation:

4. You may want to choose your own language, depending on the availability of the prescribed **Language packs**. This tab currently has Chinese, Czech, English, French, German, Italian, Japanese, Korean, Polish, Portuguese, Russian, Spanish, and Turkish as available options. You may also want to specify your custom installation path, and that can be done from the **Installation locations** tab.
5. When you have finished selecting your desired workloads and components, the installation will start. The installation time is dependent on the number of workloads and components you have selected, as well as your internet connection speed if you are not using the offline installation method we described previously.

WOW! eBook
www.wowebook.org

For more advanced scenarios, such as automating and scripting the Visual Studio 2019 installation, you can start the setup program via the Command Prompt. There are a variety of command-line parameters that can help us define what needs to be installed, and where.

The following is a list of some of the command-line parameters that are available, along with a brief description of what they do. Please go to `https://docs.microsoft.com/en-us/visualstudio/install/use-command-line-parameters-to-install-visual-studio` to find out more, including a full list of all the existing command-line parameters in addition to the ones described as follows:

Parameter	**Description**
`/AddRemoveFeatures`	This adds the features that have been selected
`/AdminFile`	This specifies a file to install silently
`/CreateAdminFile`	This specifies that you wish to generate a silent response file after your installation
`/CustomInstallPath`	This specifies the target path
`/ForceRestart`	This forces your PC to restart
`/Full`	This installs all the necessary features
`/noweb`	This disables internet searching features and downloading
`/ProductKey`	This specifies the key to be used

First steps with Visual Studio 2019

After installing Visual Studio 2019, you can now explore everything it has to offer in terms of improving developer productivity. The following is a list of some of the features that are provided.

One of the most important features of Visual Studio is IntelliSense. It helps developers be much more productive by offering features such as List Member, Parameter Info, Quick Info, and Complete Word. It has been improved in Visual Studio 2019 with some very interesting new features, such as IntelliCode, and you can now filter by type (class, namespace, or keyword) and by camelCase search.

WOW! eBook
www.wowebook.org

You can use the programming language, platform, and project type as search filters. The following options will be available for you:

- **Language**: C++, C#, Java, F#, JavaScript, Python, Query Language, TypeScript, and Visual Basic
- **Platform**: All platforms, Android, Azure, iOS, Linux, macOS, tvOS, Windows, and Xbox
- **Project Type**: All project types, cloud, console, machine learning, desktop, extensions, games, IoT, library, mobile, office, service, test, UWP, and web

It is advisable to select the best match from the list of results instead of just picking the top one as it may not always necessarily be the correct one:

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.
    }
    else
    {
        app.
    }

    app.UseS

    app.UseM
    {
        rout
                                                                    /{id?}");
    });
}
```

```
ApplicationServices
Build
Equals
GetHashCode
GetType
Map
MapWhen
New
Properties
```

```
IServiceProvider IApplicationBuilder.ApplicationServices { get; set; }
Gets or sets the IServiceProvider that provides access to the application's service container.
```

The **code refactoring** and **live code analysis** features of Visual Studio 2019 accelerate development and ensure that you have readable and maintainable code. For example, the following is an instance where you can add missing namespaces or remove unnecessary namespaces automatically:

WOW! eBook
www.wowebook.org

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Threading.Tasks;
5   using Microsoft.AspNetCore.Builder;
6   using Microsoft.AspNetCore.Hosting;
7   using Microsoft.AspNetCore.Mvc;
8   using Microsoft.Extensions.Configuration;
9   using Microsoft.Extensions.DependencyInjection;
10  using Microsoft.Extensions.Logging;
11
12  Remove Unnecessary Usings
13      public class Startup
14      {
15          public Startup(ICo
16          {
17              Configuration
18          }
19
20          public IConfigurat
21
22          // This method get                                  the container
23          public void Config
24          {
25              services.AddMvc();
26          }
27
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
...
Preview changes
Fix all occurrences in: Document | Project | Solution
```

The following is an example of a code refactoring suggestion, which shows up as a light bulb in this instance:

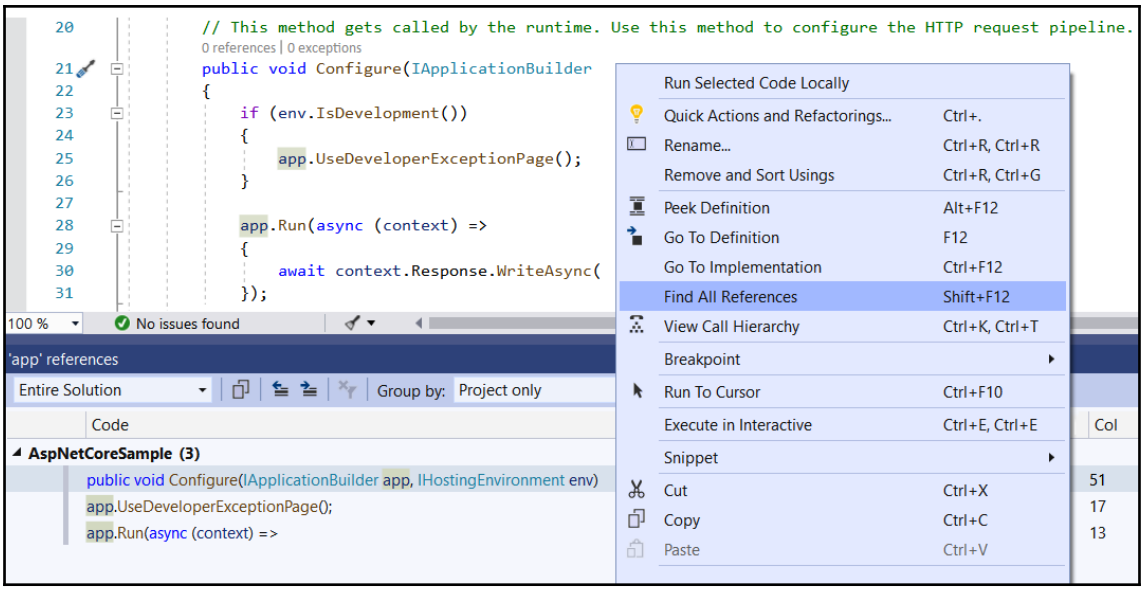
```
23          // This method gets called by the runtime. Use this method to add services to the container.
24          public void ConfigureServices(IServiceCollection services)
25          {
2           services.AddMvc();
2
2           Use expression body for methods
29          // This method                                      pipeline.
30          public void Con
31          {
32              if (env.IsD
33              {
34                  app.Use
35                  app.Use
36              }
```

```
...
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
public void ConfigureServices(IServiceCollection services) => services.AddMvc();
...
Preview changes
```

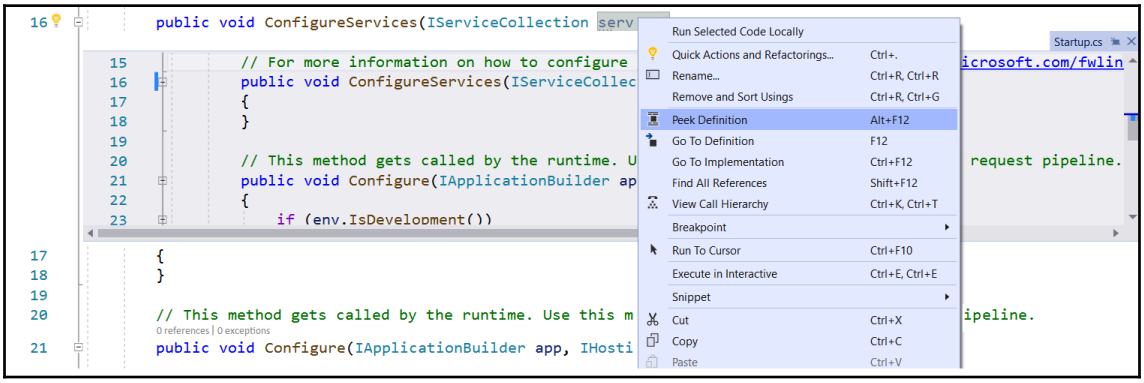
With Visual Studio 2019, you will find that it includes features that were only previously available as enhancements with external plugins such as ReSharper. An example of this is the fact that you can now convert `foreach` loops into more concise and performant LINQ statements.

WOW! eBook
www.wowebook.org

As its name suggests, the **Find All References** feature allows a developer to easily and quickly find all references for a method or an object. Coloring, grouping, and peek preview functionality aid you visually so that you can navigate within your code and really understand it:

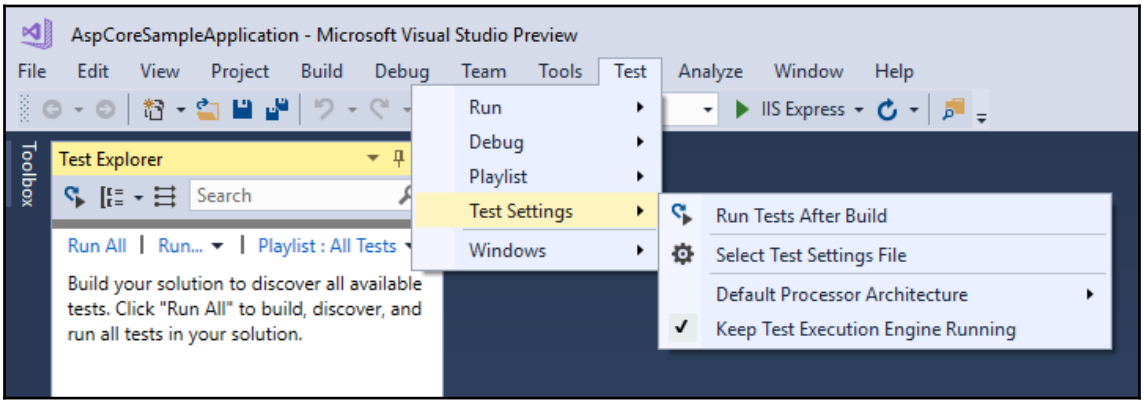


The **Peek Definition** and **Go To Definition** features allow you to examine the definition of a method, interface, or class, within a popup window, without changing the current window, or by directly opening the file containing the source code with the requested definition. The **Go To Implementation** feature does the same, but navigates to the implementation instead:



WOW! eBook
www.wowebook.org

Another important feature is Live Unit Testing. You will need Visual Studio 2019 Enterprise Edition to use it. It allows you to automatically run unit tests in the background after each modification or compilation of your code. It can be configured and activated in the **Test Settings** menu. From here, you can set, for instance, the number of test processes, the maximum duration for each test, and the maximum memory consumption:



There are many more interesting and exciting features in Visual Studio 2019, and we invite you to visit the official Visual Studio web page at `https://docs.microsoft.com/en-us/visualstudio/welcome-to-visual-studio` if you want to find out more. It is key for a developer to know their developer IDE as best they can and to familiarize themselves with a lot of its features so that they can do their job better and faster. So, do take some time to look at this before you start developing your applications.

Creating your first ASP.NET Core 3 application in Visual Studio 2019

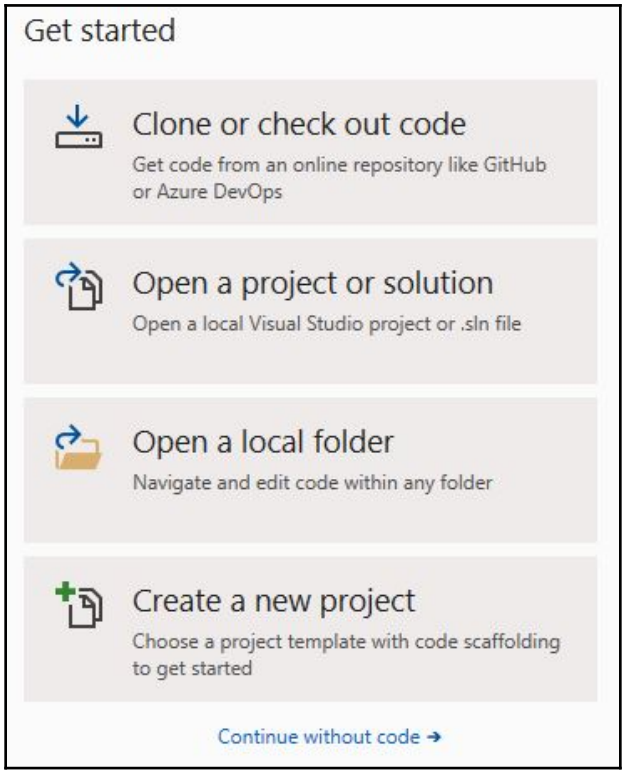
You have patiently read the previous chapters, understood what you will be learning about by reading this book, and prepared your developer machine. Now, you are ready to create your first sample application.

Follow these instructions to create your first ASP.NET Core 3 sample web application:

1. If you haven't installed the .NET Core 3 SDK yet, then download and install .NET Core 3 from `https://dotnet.microsoft.com/download/dotnet-core/3.0`.

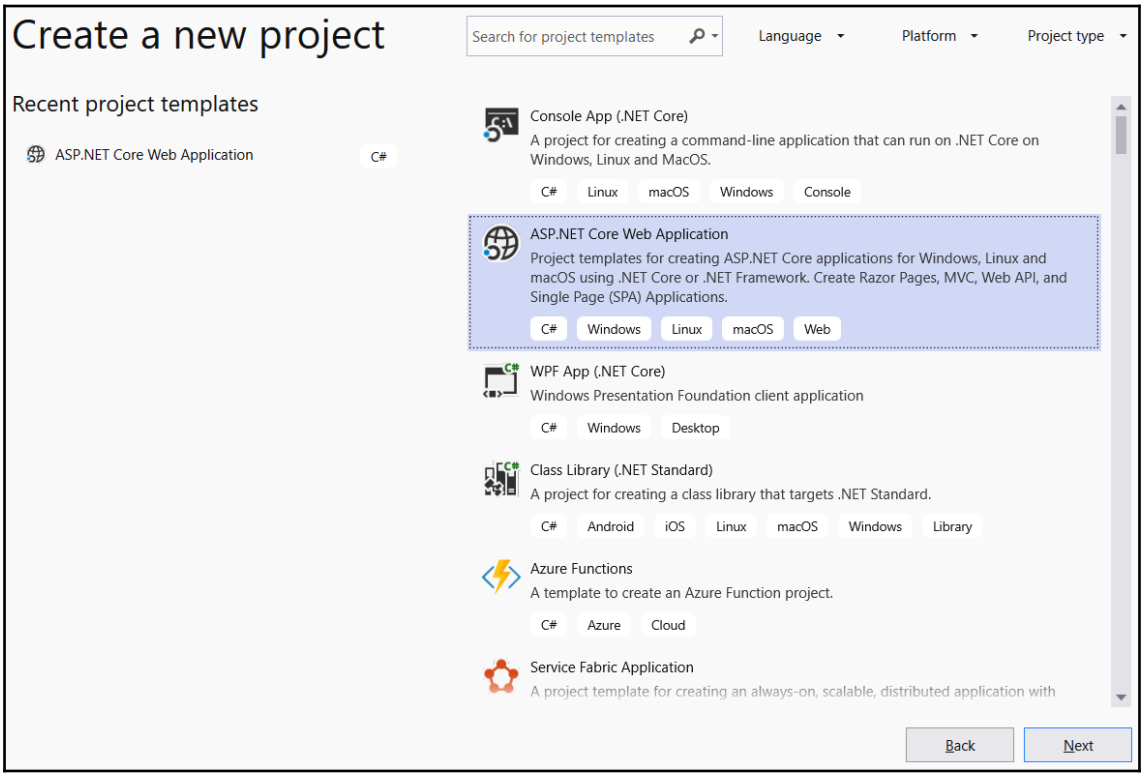
WOW! eBook
www.wowebook.org

2. Start Visual Studio 2019. You will be presented with a page that allows you to clone or checkout code or open an existing project. Here, we are interested in creating a new project:



WOW! eBook
www.wowebook.org

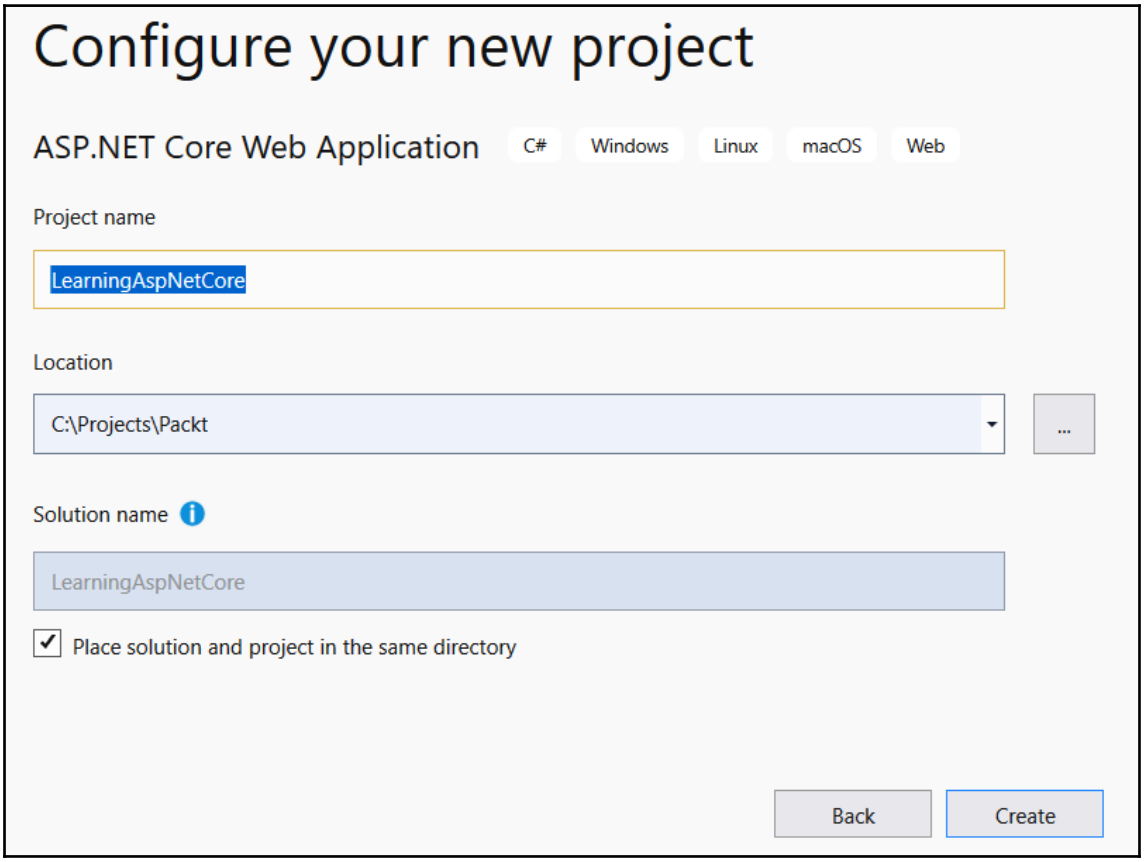
3. The same **Create a new project** page pops up if you click on **Create a new project** or **Continue without code** and then **File | New | Project**:



Select **ASP.NET Core Web Application** and click **Next**.

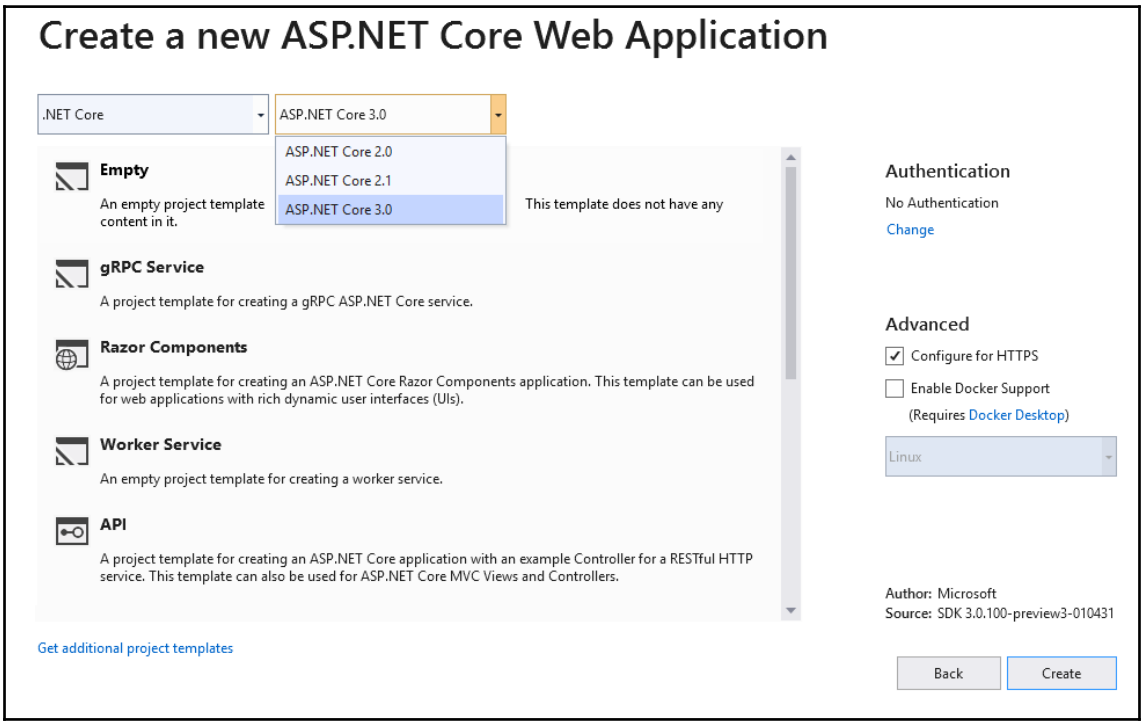
WOW! eBook
www.wowebook.org

4. After selecting a project template, that is, **Visual C# | .NET Core | ASP.NET Core Web Application (.NET Core)**, and clicking **Next**, you can name your project and configure its location:



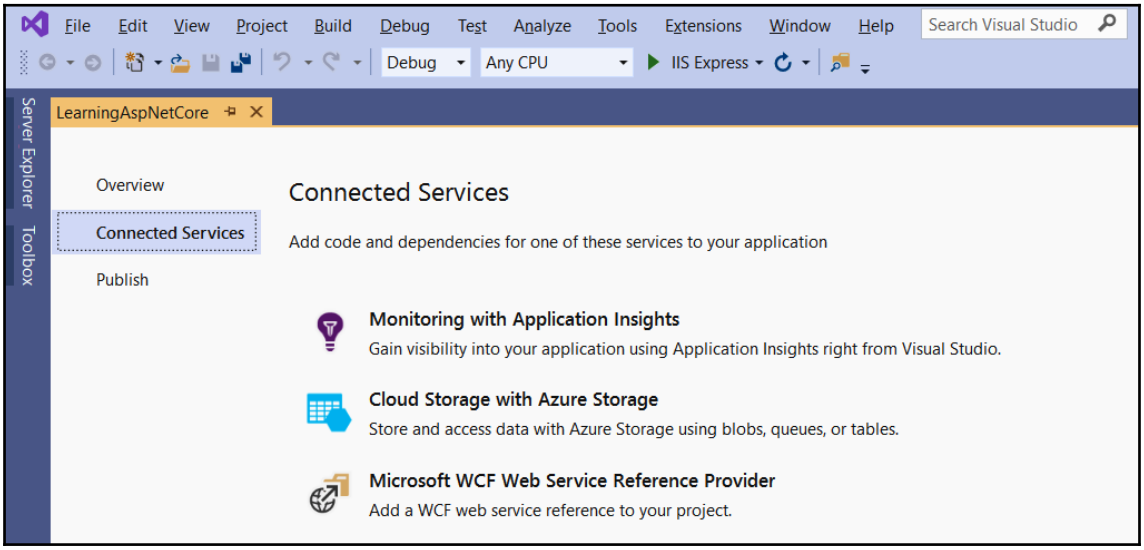
WOW! eBook
www.wowebook.org

5. You are now able to select your specific web application type, which is the ASP.NET Core version. Please note that, if we select **ASP.NET Core 3.0**, then we only have .NET Core available on the left, but for all the others, we also have .NET Framework available for selection as well. Scroll down and select **Web Application** and leave the **Enable Docker Support**, and **Configure for HTTPS** options unchecked. Also, leave **Authentication** set to **No Authentication**:



WOW! eBook
www.wowebook.org

6. After the sample application project has been generated, a project start page will be displayed. Here, you can configure additional options, such as connected services (Application Insights, and more) and publishing targets (**Microsoft Azure App Services**, **IIS**, **FTP**, **Folder**, and more). Leave everything unchanged:



7. Now, you can start your application by pressing *F5* or clicking on **Debug | Start Debugging**.

Creating your first ASP.NET Core 3 application via the command line

In the previous section, you learned how to create your first ASP.NET Core 3 web application with Visual Studio 2019. This should be the preferred method for most developers.

However, if you prefer using the command line or Visual Studio Code, which we are going to introduce a little later on in this book, then using Visual Studio 2019 is not really an option. Luckily, .NET Core and ASP.NET Core 3 provide full support for the command line. This may even be your only option on other operating systems such as Linux or macOS. The same command-line instructions work on all the different operating systems, so once you get used to them, you can work on any environment.

WOW! eBook
www.wowebook.org

Now, let's learn how to create our first sample application using the Windows command line:

2. If you haven't installed the .NET Core 3 SDK yet, then download and install .NET Core 3 from `https://dotnet.microsoft.com/download/dotnet-core/3.0`.
3. Create a folder for your sample application called `mkdir aspnetcoresample`.
4. Move into the new folder: `cd aspnetcoresample`.
5. Create a new web application based on the empty ASP.NET Core 3 web application template called `dotnet new web`.

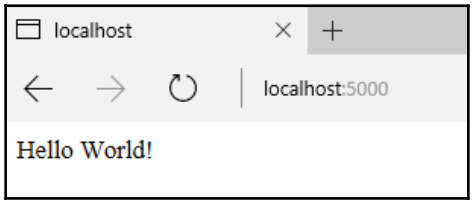
> Previous versions of .NET Core required an additional `-t` parameter for choosing a template (`dotnet new -t web`). If you get an error when executing `dotnet new web`, it is a good indication that you need to install .NET Core 2.0.
> Note that you can verify your .NET version by entering `dotnet` (with no parameters) if you are not sure about your environment since it will display the current .NET Core version.

5. Run the sample application by executing `dotnet run`:

```
Administrator: Command Prompt - dotnet run
C:\Projects\Packt\aspnetcoresample>dotnet new web
The template "ASP.NET Core Empty" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Projects\Packt\aspnetcoresample\aspnetcoresample.csproj...
  Restore completed in 97.2 ms for C:\Projects\Packt\aspnetcoresample\aspnetcoresample.csproj.

Restore succeeded.


C:\Projects\Packt\aspnetcoresample>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Projects\Packt\aspnetcoresample
```


WOW! eBook
www.wowebook.org

6. Open a browser and go to `http://localhost:5000`. If everything worked correctly, you should see a **Hello World!** page:



In this section, you've learned how to create your first sample application by using Visual Studio 2019 or the command line, and you learned how to use Visual Studio Code and how it helps you when building an ASP.NET Core 3 application on Linux or macOS.

Now that you have installed Visual Studio 2019 and have your first application up and running, you need to know what to do when you encounter errors in your application. This is a matter of when, not if. Do not despair if you do get errors when developing applications – this happens even to the most experienced. Luckily, Visual Studio helps us to diagnose errors. We will look at debugging with Visual Studio 2019 in the next section.

Basic debugging with Visual Studio 2019

Whenever we write the logic for software applications, there are times when we manage to achieve the intended functionality outright without any problems and errors. While it is often desirable to get it right the first time, this will most definitely not always be the case for most software developers.

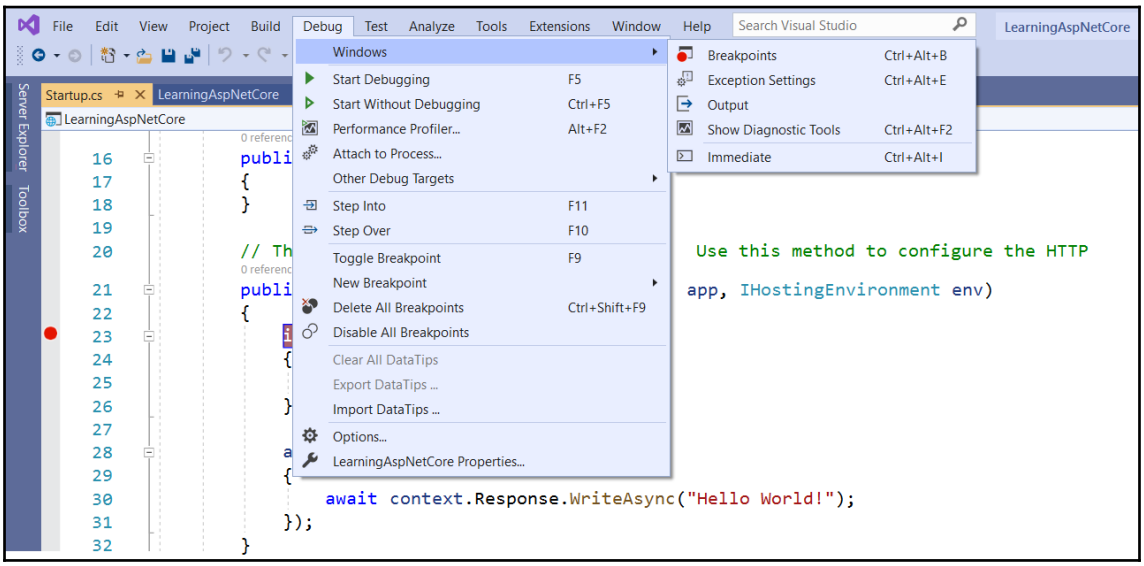
We may have a situation where our code has compiled successfully but we find out that we don't have the output we wanted, or we may get compile-time or runtime errors. In this case, it is quite helpful for a developer to find the errors before anything is discovered after the software has been released.

Errors within an application are syntactical or semantical, in which you as a developer either haven't followed the prescribed language syntax or are not making logical sense. These kinds of error are easier to find. The Visual Studio IDE will help you to catch most of these development errors while downright refusing to compile or throw an exception after you run your applications in debug mode.

WOW! eBook
www.wowebook.org

There is another set of application errors that arise from not being able to produce intended behaviors, even though we have done our best to code against the prescribed functionality. These are best counter-checked by writing unit tests and running them against our code. Some professionals within the software development industry advocate **test-driven development** (**TDD**), in which the tests are written before any functionality is written, while others may consider this a duplication of effort and are not so bullish about this. Whatever the case, unit tests are quite important in an application and we will spend some time later in the book, when we provide practical examples, demonstrating their worth.

The following screenshot is a snapshot of the debugging functionality that's available in Visual Studio 2019:



We will not be going through all of the debugging functionality that's available through the **Debug** menu shown in the preceding screenshot because this would take a whole book of its own, but it is well worth exploring as a developer. Knowing the basics of debugging will save you a lot of time as a developer. In the next section, we will explain a few of the most important debug items that are available to you in Visual Studio 2019.

WOW! eBook
www.wowebook.org

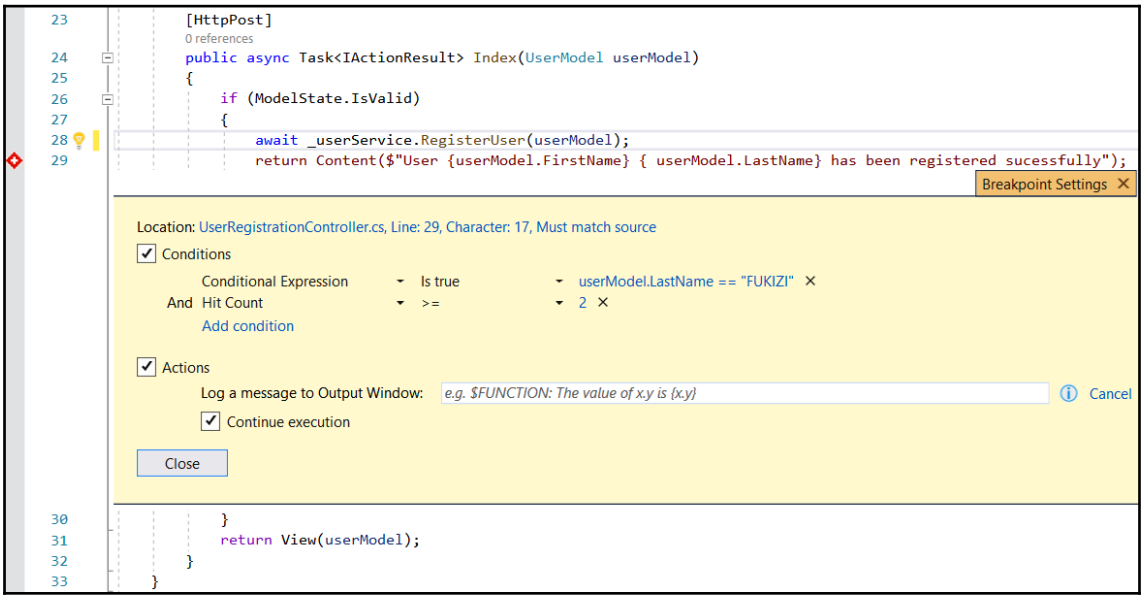
Breakpoints

A **breakpoint** is one of the most important tools in debugging and is represented as a red dot in the preceding screenshot. When you start an application in debugging mode through the **Debug** menu, it runs sequentially through your code until it hits a breakpoint you have placed on any point in your code base, from which point you can choose to step into a code statement and inspect it, step over it, or step out of a set of statements.

A breakpoint will give you access to the actual values of your objects at that specific instance, which will help you inspect the behavior of your program at a point of concern, and therefore help you in troubleshooting whatever problem you may be having.

A program can have as many breakpoints as needed, and you are able to enable or disable these breakpoints in bulk.

There will be times when you may need Visual Studio to break only in a specific condition, such as when a property changes or some condition becomes true. Luckily, Visual Studio provides an out-of-the-box solution for this scenario. You are able to set what is normally referred to as a **conditional breakpoint** by right-clicking any normal breakpoint and then selecting **Conditions**. When you click on it, a pop up will appear where you can set your condition, as follows:



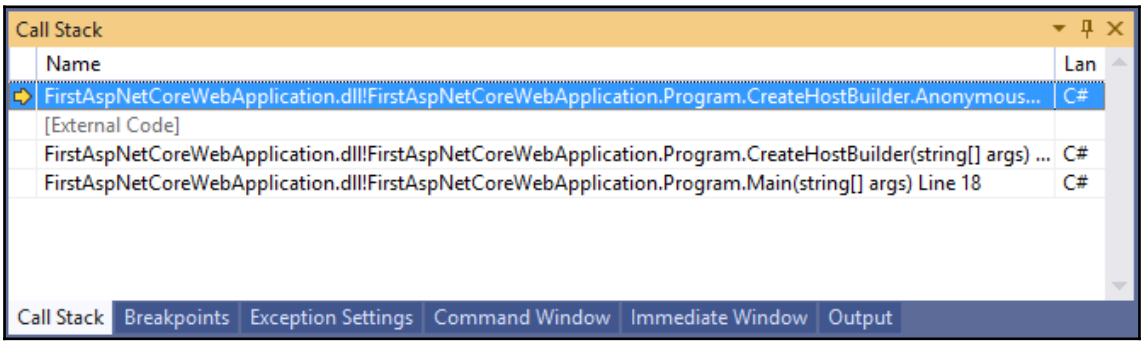
WOW! eBook
www.wowebook.org

The preceding screenshot is just a hypothetical example which shows how we can look at the `LastName` in a user model and check whether it is equal to our chosen string, `"FUKIZI"`. In the preceding example, we set another condition called `Hit Count >= 2`, which means that our conditional breakpoint will only trigger after it has been hit two or more times.

We could also set an action to output a message or choose whether we want the execution to continue. Sometimes, we may need to look back on what was really happening before a certain point is reached. For this, the call stack comes in handy.

Call stack

A call stack will give you a snapshot history of the calls that have been made by your program to get you to the point where you are in debugging mode:

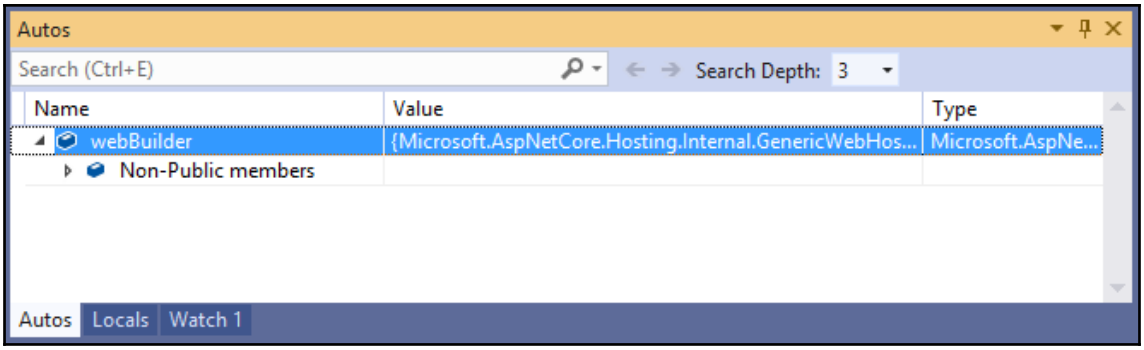


This is quite helpful if you wish to inspect what your code has been doing in its immediate history, and may help you to locate where the problems in your code may have originated.

WOW! eBook
www.wowebook.org

Autos, Locals, and Watch Panes

Apart from hovering over your breakpoints to see what is contained in a variable, the Autos window is the next most important tool when debugging as it gives you a snapshot of variable contents in a more persistent way on the screen. You can add a watch to a specific variable of interest and it will appear in the watch pane. You can manipulate and modify the values of variables as a way of testing what should have been contained or passed on in a variable or object to try and get an intended behavior before deciding where to change your problematic code when troubleshooting a problem:



Apart from the aforementioned tools, and others that are beyond the scope of this book, it must be said that debugging is a skill that gets cemented with time and experience. The more applications you write and debug, the more intuitive and natural it will become for you to discover where your application is misbehaving.

We have covered some ground so far using Visual Studio 2019 as our integrated development environment. It is such a great tool and is loved by many, but there are also other tools with great features that can be used to develop ASP.NET Core 3 applications. One such tool that is loved and used by many is Visual Studio Code, which we will introduce in the next section.

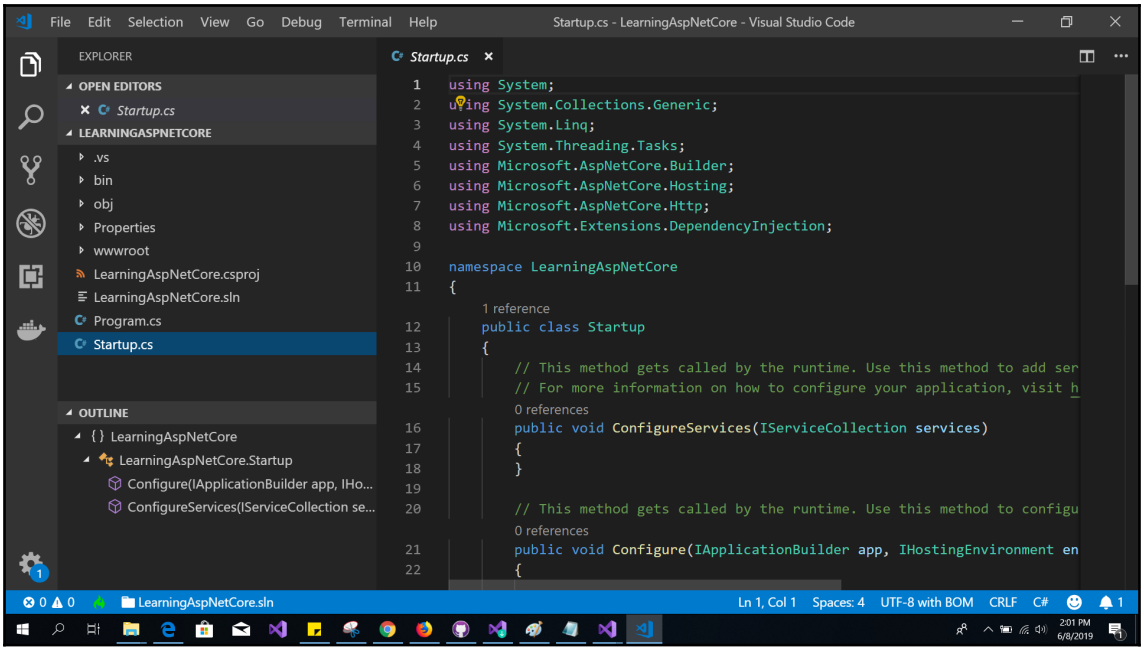
Visual Studio Code as a development environment

Visual Studio Code is a lightweight and powerful cross-platform development environment for Windows, Linux, and macOS.

WOW! eBook
www.wowebook.org

You can use a wide range of programming languages such as JavaScript, TypeScript, and Node.js, as well as C++, C#, Python, PHP, Go, and .NET Core and Unity runtimes via language and runtime extensions.

It comes with a streamlined, clean, and very efficient user interface. There's a file and folder explorer on the left and a source code editor on the right, which shows the contents of the files you have opened and are currently working on:



The user interface consists of the following areas:

- **Activity bar**: Provides several different views and additional context-specific indicators, such as outgoing code changes when Git is enabled.
- **Sidebar**: Contains a file and folder explorer for working on your projects.
- **Editor groups**: This is the main area for working with your code and navigating within it. Up to three source code editor windows can be opened side by side at the same time.
- **Panels**: Displays panels with output or debug information, errors, and warnings, or an integrated Terminal.
- **Status bar**: Supplies additional information concerning projects and files you have edited.

WOW! eBook
www.wowebook.org

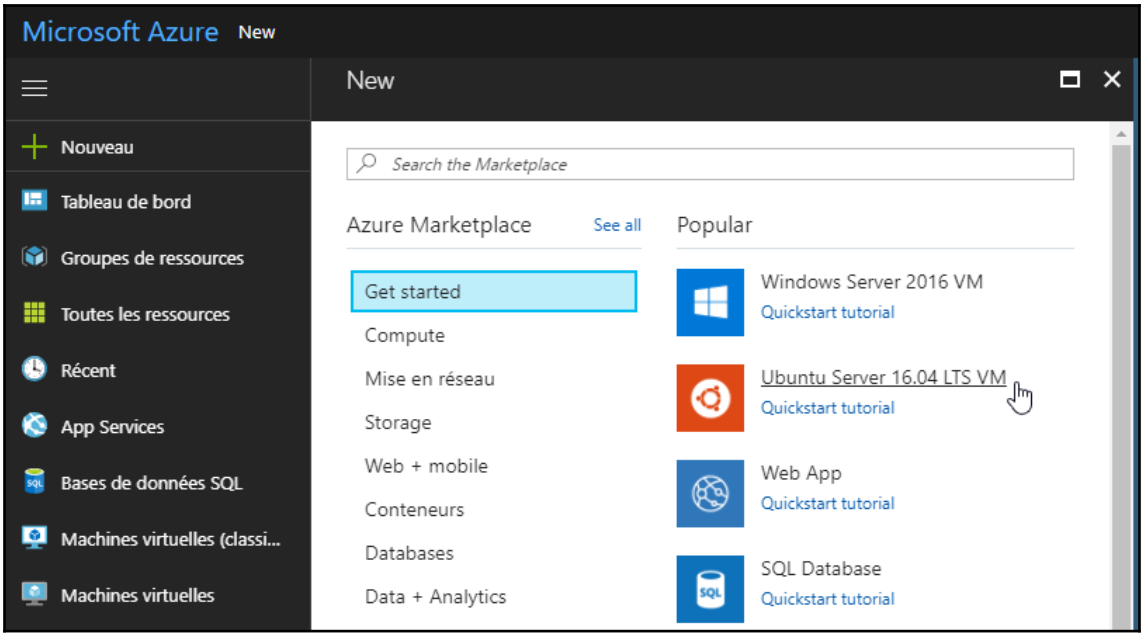
Please go to `https://code.visualstudio.com/docs` for additional information on Visual Studio Code and its capacities and functionalities. It will be our primary choice for illustrating how to build ASP.NET Core 3 applications on Linux.

How to install Visual Studio Code on Linux

In this section, we are going to explain how easy and fast it is to install Visual Studio Code on Linux. One of the most popular Linux distributions, Ubuntu 16.04, will serve as an example.

If you do not have a physical or virtual installation of Linux Ubuntu available, you can easily install it in Azure to try out Visual Studio Code and understand the various ASP.NET Core 3 examples. By doing this, you can connect via the Microsoft Remote Desktop app.

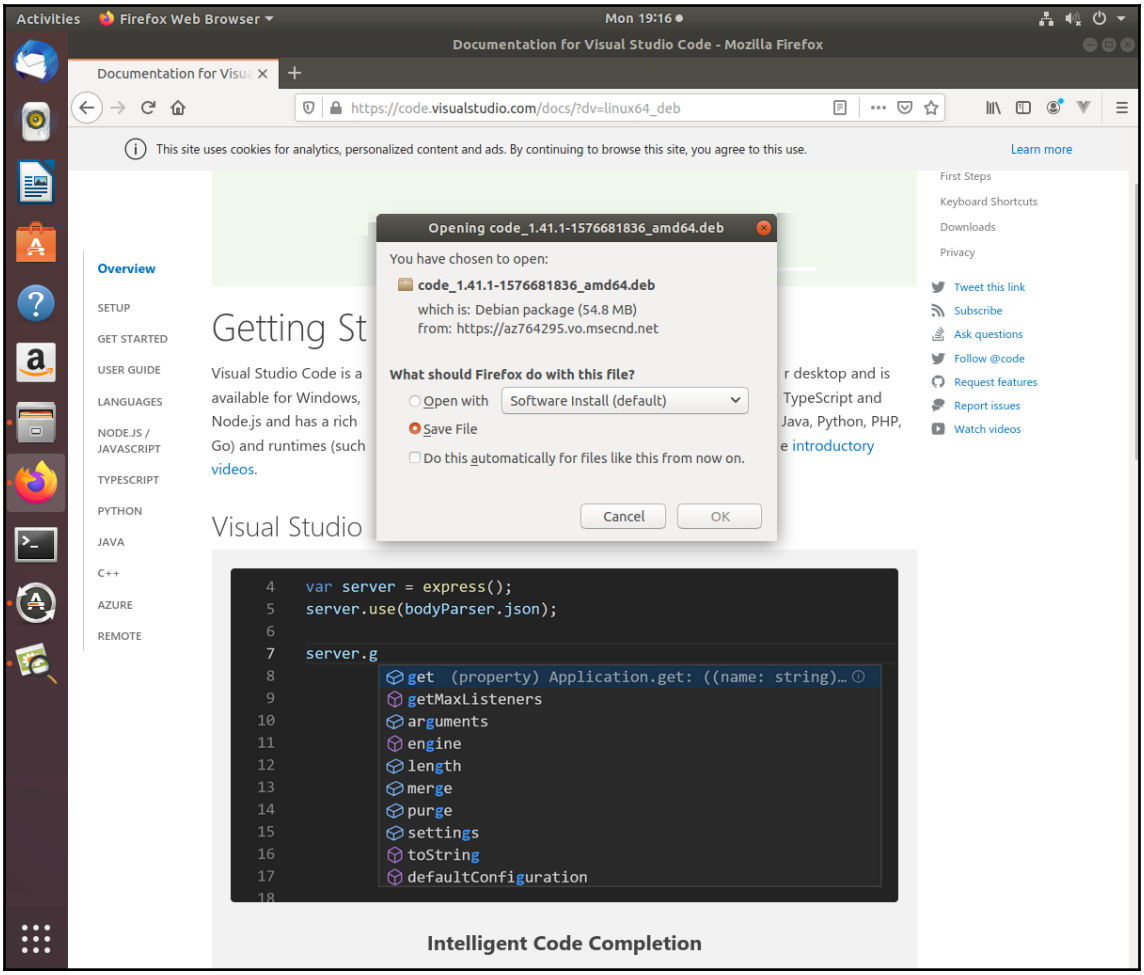
In this case, select the Linux Ubuntu 18.04 LTS image from the Azure Marketplace and create a new Linux Ubuntu VM in Azure. Leave all of the default options and configure it to allow remote desktop connections (install a compatible desktop, install **xrdp**, open port `3389`, and more):



WOW! eBook
www.wowebook.org

Let's learn how to install Visual Studio Code on Linux Ubuntu:

1. First, download the Linux Ubuntu installation `.deb` package (64-bit) from `https://go.microsoft.com/fwlink/?LinkID=760868`



2. Open a new Terminal window in Ubuntu.
3. Install the downloaded package via `sudo dpkg -i <file>.deb`.
4. Then, enter `sudo apt-get install -f`.
5. Set Visual Studio Code as your default text file editor by typing in the `xdg-mime default code.desktop text/plain` command.

WOW! eBook
www.wowebook.org

The installation will begin and automatically install the APT repository and signing key to enable automatic package updates, as well as Visual Studio Code:

```
jason@ubuntu-asp-core: ~/Downloads
jason@ubuntu-asp-core:~$ ls
Desktop    Downloads         Music     Public     Videos
Documents  examples.desktop  Pictures  Templates
jason@ubuntu-asp-core:~$ cd Downloads/
jason@ubuntu-asp-core:~/Downloads$ sudo dpkg -i code_1.13.1-1497464373_amd64.deb

[sudo] password for jason:
Selecting previously unselected package code.
(Reading database ... 174260 files and directories currently installed.)
Preparing to unpack code_1.13.1-1497464373_amd64.deb ...
Unpacking code (1.13.1-1497464373) ...
Setting up code (1.13.1-1497464373) ...
Processing triggers for gnome-menus (3.13.3-6ubuntu3.1) ...
Processing triggers for desktop-file-utils (0.22-1ubuntu5) ...
Processing triggers for bamfdaemon (0.5.3~bzr0+16.04.20160824-0ubuntu1) ...
Rebuilding /usr/share/applications/bamf-2.index...
Processing triggers for mime-support (3.59ubuntu1) ...
jason@ubuntu-asp-core:~/Downloads$ sudo apt-get install -f
Reading package lists... Done
Building dependency tree
Reading state information... Done
0 upgraded, 0 newly installed, 0 to remove and 241 not upgraded.
jason@ubuntu-asp-core:~/Downloads$ xdg-mime default code.desktop text/plain
jason@ubuntu-asp-core:~/Downloads$
```

You can also manually install the repository and signing key, update the package cache, and start the Visual Studio Code package installation, as follows:

1. Open a new Terminal window in Ubuntu:

```
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --
 dearmor>microsoft.gpg

sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg

sudo sh -c 'echo "deb [arch=amd64]
 https://packages.microsoft.com/repos/vscode stable main" >
  /etc/apt/sources.list.d/vscode.list'

sudo apt-get update

sudo apt-get install code
```

2. Set Visual Studio Code as your default text file editor by typing in the `xdg-mime default code.desktop text/plain` command.

WOW! eBook
www.wowebook.org

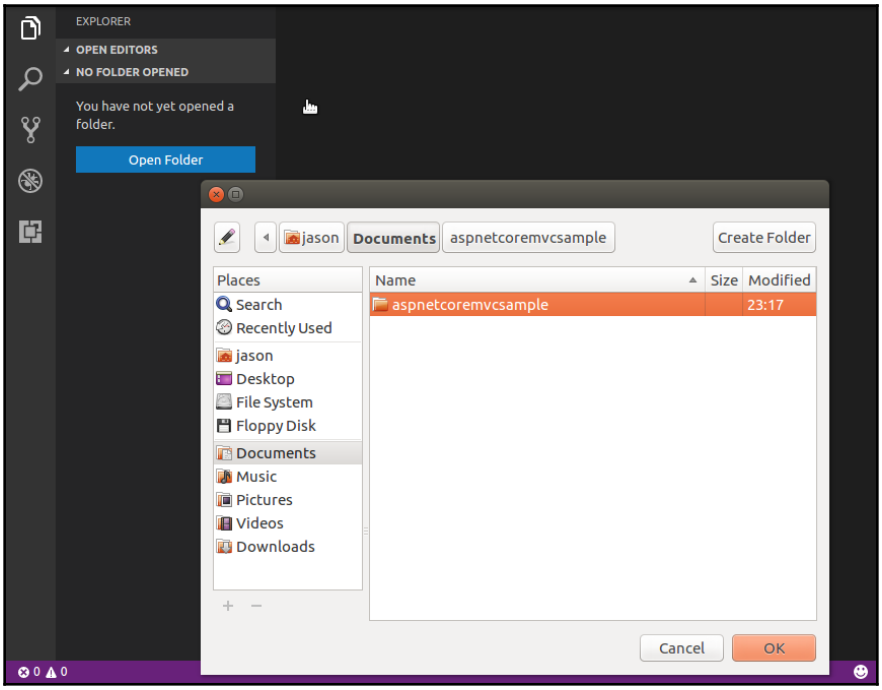
For more information on how to install Visual Studio Code on other Linux distributions, such as RHEL, Fedora, CentOS, openSUSE, SLE, and others, please go to `https://code.visualstudio.com/docs/setup/linux`.

At this point, we have our environment ready and can create our first ASP.NET Core application in our new Linux environment.

Creating your first ASP.NET Core 3 application in Visual Studio Code

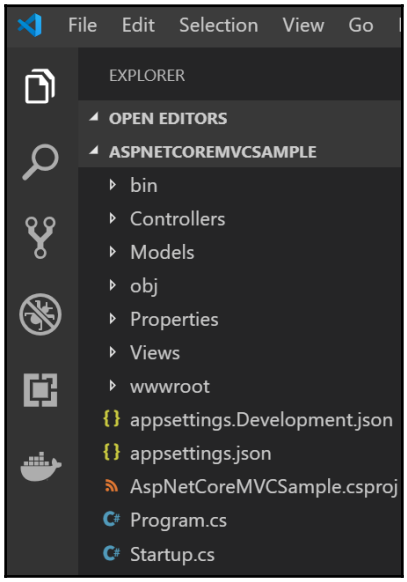
Now, you will learn how to initialize your first ASP.NET Core 3 application using the built-in Visual Studio Code Terminal window. Then, you are going to install all of the necessary extensions so that you can run and debug it:

1. Start Visual Studio Code.
2. Click on **Open Folder** and click on **Create Folder**. Name the folder `aspnetcoremvcsample` and click on **OK**:

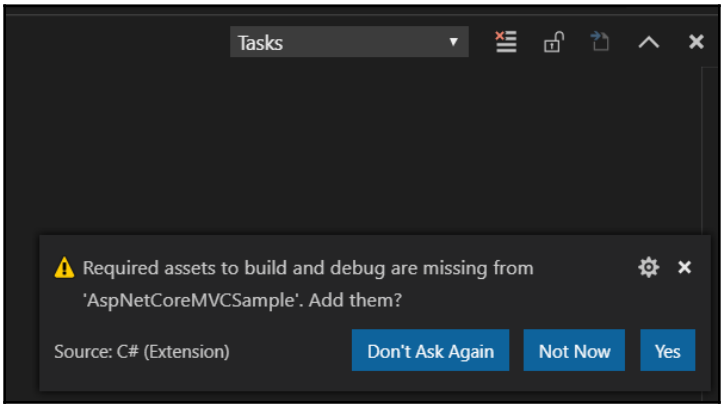


WOW! eBook
www.wowebook.org

3. Display the integrated Terminal window via **View** | **Integrated Terminal** and initialize a new ASP.NET Core 3 MVC project by entering `dotnet new mvc`:

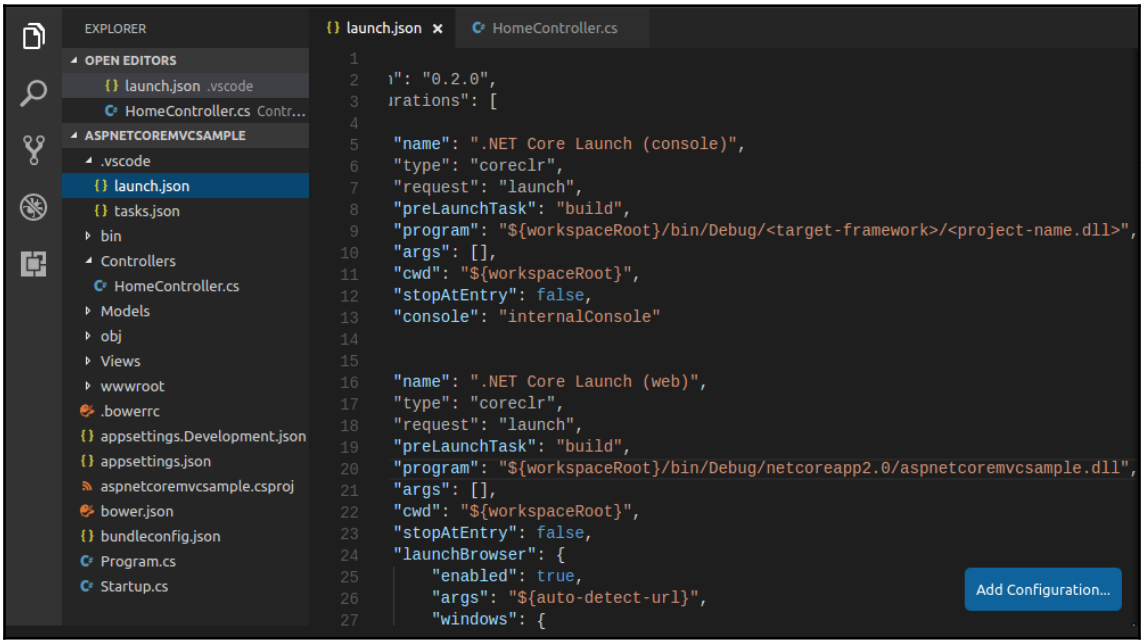


4. When opening a C# file, you will be asked to install additional project dependencies and Visual Studio Code extensions. You need to do this to be able to build, run, and debug your application in the steps that follow:



WOW! eBook
www.wowebook.org

5. Modify the `launch.json` file in the `.vscode` folder and set the debugger to **.NET Core Launch (web)**:



6. Set a breakpoint anywhere in the code and start debugging by either pressing *F5* or clicking on the green flash in the debugging viewlet. Try hitting the breakpoint; everything should work correctly.

Creating your first ASP.NET Core 3 application in Linux

To create and run your first sample application using only the Terminal window in Linux, follow these steps:

1. If you haven't installed the .NET Core 3 SDK yet, then download and install it from `https://dotnet.microsoft.com/download/dotnet-core/3.0` for your Linux distribution. The following is an example of how to do that for Ubuntu:

```
sudosh -c 'echo "deb [arch=amd64]
 https://apt-mo.trafficmanager.net/repos/dotnet-release/
  xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
```


WOW! eBook
www.wowebook.org

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
 --recv-keys 417A0893
sudo apt-get update
sudo apt-get install dotnet-sdk-2.0.0-preview2-006497
```

2. Create a folder for your sample application called `aspnetcoremvcsample`: `mkdir ~/Documents/aspnetcoremvcsample`.
3. Move into the new folder, that is, `cd ~/Documents/aspnetcoremvcsample`.
4. Create a new web application based on the ASP.NET Core 3 MVC web application template and called `dotnet new mvc`:

```
jason@ubuntu-asp-core: ~/Documents/aspnetcoremvcsample
jason@ubuntu-asp-core:~/Documents$ mkdir ~/Documents/aspnetcoremvcsample
jason@ubuntu-asp-core:~/Documents$ cd ~/Documents/aspnetcoremvcsample/
jason@ubuntu-asp-core:~/Documents/aspnetcoremvcsample$ dotnet new mvc
The template "ASP.NET Core Web App (Model-View-Controller)" was created success
fully.
This template contains technologies from parties other than Microsoft, see http
s://aka.ms/template-3pn for details.

Processing post-creation actions...
Running 'dotnet restore' on /home/jason/Documents/aspnetcoremvcsample/aspnetcor
emvcsample.csproj...
  Restoring packages for /home/jason/Documents/aspnetcoremvcsample/aspnetcoremv
csample.csproj...
  Installing Microsoft.IdentityModel.Logging 1.1.3.
  Installing System.Collections.Immutable 1.3.0.
  Installing System.Xml.XmlSerializer 4.0.11.
  Installing System.Xml.XmlDocument 4.0.1.
  Installing runtime.osx.10.10-x64.runtime.native.System.Security.Cryptography.
Apple 4.3.0.
  Installing System.Threading.Tasks.Extensions 4.0.0.
  Installing System.Security.Cryptography.Csp 4.0.0.
  Installing Microsoft.NETCore.App 2.0.0-preview2-25407-01.
  Installing System.Security.Cryptography.Cng 4.2.0.
  Installing System.Security.Cryptography.OpenSsl 4.0.0.
```


WOW! eBook
www.wowebook.org

5. Run the sample application by executing `dotnet run`:

```
jason@ubuntu-asp-core: ~/Documents/aspnetcoremvcsample
  Installing NuGet.Frameworks 4.0.0.
  Installing Microsoft.Build 15.3.0-preview-000388-01.
  Installing System.Diagnostics.TraceSource 4.0.0.
  Installing Microsoft.Build.Framework 15.3.0-preview-000388-01.
  Installing Microsoft.Build.Tasks.Core 15.3.0-preview-000388-01.
  Installing Microsoft.Build.Utilities.Core 15.3.0-preview-000388-01.
  Installing System.Text.Encoding.CodePages 4.0.1.
  Installing System.Diagnostics.Contracts 4.0.1.
  Installing System.Collections.Immutable 1.2.0.
  Installing System.Collections.NonGeneric 4.0.1.
  Installing System.Diagnostics.FileVersionInfo 4.0.0.
  Installing System.Diagnostics.Process 4.1.0.
  Restore completed in 2.06 sec for /home/jason/Documents/aspnetcoremvcsample/a
spnetcoremvcsample.csproj.


Restore succeeded.

jason@ubuntu-asp-core:~/Documents/aspnetcoremvcsample$ dotnet run
Hosting environment: Production
Content root path: /home/jason/Documents/aspnetcoremvcsample
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

6. Open a browser and go to `http://localhost:5000`

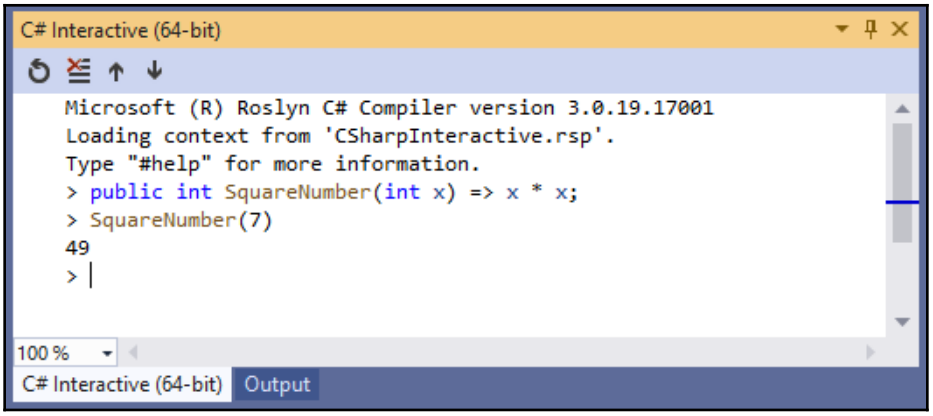
In this section, we've seen our first application running on different operating systems and briefly explored the tools that are available within the Integrated Development Environment. We will look at C# Interactive in the next section. There are also other external tools that can help make development as painless as possible. One such tool is LINQPad, which we will introduce in the next section as well.

Introduction to the C# Interactive and LINQPad tools

C# Interactive is a **read-eval-print-loop** (**REPL**) tool that is commonly ignored by many developers who use Visual Studio, but it has amazing functionality that will help you experiment with a notion quickly before you implement it. You can use it to play around with C# language features and any .NET technologies. For example, if you are not too sure how a certain ASP.NET Core 3 feature works, you can reference it in the pane and, interact with it, and view the output immediately. This makes it much easier and faster than experimenting with a full application.

WOW! eBook
www.wowebook.org

In the following simple example, we'll declare a function called `SquareNumber`, which takes in an integer, `x`, multiplies it by itself, and gives us the answer when it's called in the next line:



With its high interactivity, you can write and test scripts and get an output immediately, which you can then use by just copying your code into the pane and pasting it into a script file that you name with the `.csx` extension; you can then run it through the Developer Command Prompt with the `csi` keyword.

You can also use C# Interactive to learn about external APIs that you can play around with interactively, but this is beyond the scope of this book.

In `Chapter 9`, *Accessing Data Using Entity Framework Core 3*, we will use LINQPad to demonstrate how to work with and debug LINQ queries. This is the right moment to introduce LINQPad, which can be downloaded from `https://www.linqpad.net/`.

Please note that LINQPad 5 supports .NET Framework and that LINQPad 6 supports the .NET Core 3.0 SDK.

LINQPad can be used for many other things, including as a testing ground for statements, expressions, and scripts in Microsoft's major languages (C#, F# and VB.NET), but our interest lies in using it to help us understand **Language-Integrated Queries** (**LINQs**) when we tackle that subject in `Chapter 9`, *Accessing Data Using Entity Framework Core 3*.

WOW! eBook
www.wowebook.org

Summary

In this chapter, you have learned how to set up your development environment so that you can work with ASP.NET Core 3 by installing either Visual Studio 2019 or Visual Studio Code.

Then, you then created your first ASP.NET Core 3 web application in both development environments and built a project in Linux to showcase their cross-platform capabilities.

In the next chapter, we will talk about how to set up a continuous integration pipeline by using Visual Studio Azure DevOps, including work items, Git branches, and build and release pipelines.

WOW! eBook
www.wowebook.org

3
Continuous Integration Pipeline in Azure DevOps

Building great applications is not a trivial task. On the contrary, it is a difficult and complex endeavor in which many professionals need to work together efficiently to create applications that correspond to high end user expectations.

Today, everything moves very fast, and **time to market** is very important for success. This chapter is going to introduce methods, processes, and tools to help you optimize your development processes, thus building high-quality software with short release cycles.

Traditionally, building software involves planning whole projects from beginning to end, writing detailed specifications, developing and testing (often in a rush), while hoping that everything will work as expected (as illustrated by the V-model approach).

Sometimes, this approach works, and sometimes, it does not. When it does not work, developers implement features while only testing manually, with the objective of adding unit tests later. Then, at the end of the project, they have to speed up to assure on-time delivery, and hence often run out of time.

This leads to projects with significant technical, functional, and quality flaws, with a high number of bugs and tremendous maintenance effort, resulting in long release cycles. In the worst-case scenario, end users will not like the delivered features; hence, the final product could be considered to be a complete failure. There is a better way of doing things, and this is something people have been talking about for some time now, and that you have surely already heard of—agile methodologies!

Agile methodologies, when combined with **continuous integration** (**CI**) and **continuous deployment** (**CD**), provide solutions for building better software with a fast time to market, lower maintenance costs, better overall quality, and higher customer satisfaction.

WOW! eBook
www.wowebook.org

While this book is not about agile methodologies as such, we recommend familiarizing yourself with the subject, and we are going to explain all of the tools and processes related to it.

In this chapter, we will cover the following topics:

- CI, CD, and build and release pipelines
- Using Azure DevOps for CI and CD
- Creating a free Azure DevOps subscription and your first Azure DevOps project
- Organizing your work via work items
- Using Git as a **version control system** (**VCS**)
- Creating an Azure DevOps build pipeline
- Creating an Azure DevOps release pipeline

Technical requirements

In order to go through this chapter, the following is required:

- Microsoft account
- Azure DevOps subscription
- GitHub account

CI, CD, and build and release pipelines

When using CI, development teams write code. After a code review, this gets integrated into a VCS, from where it is built and tested automatically. This normally happens multiple times a day. Thus, a development team can detect problems and bugs quickly, and fix them as early as possible, enabling what is commonly called **fail fast**.

CD is a natural extension of CI since it assures that every application modification, after being built and tested, is releasable. Development, testing, staging, and production systems are automatically upgraded through CD.

A pipeline defines a complete development and release workflow. It contains all of the steps required for conception, development, quality assurance, and testing, until delivery of the final product. It includes CI and CD processes for building high-quality applications in an industrialized way.

WOW! eBook
www.wowebook.org

Note that you can separate your development process into two different pipelines—a build pipeline and a release pipeline—or have only one single pipeline that does it all, depending on your specific needs.

There are various technologies and tools to help you implement an efficient, productive, fully automated, and industrialized software development process based on CI and CD. We are going to use Microsoft Azure DevOps in the following examples:

Using Azure DevOps for CI and CD

If you need to collaboratively work together and share code, plan and manage your user stories and development tasks, track the progress of your features and bugs, all in an agile environment, then Azure DevOps is perhaps one of the best solutions you can find in the cloud.

It supports many different programming languages (C#, Java, JavaScript, and more), various development tools (Visual Studio, Eclipse, and more), and is scalable to any team size.

Additionally, it is free of charge for up to five users in a private team project, which is very helpful for trying out the examples shown in this book.

Azure DevOps provides the following main features:

- **Work items and the Kanban board**: Plan and assign work and tasks.
- **Source code management**: Share code in a VCS.
- **Testing**: Create and execute test plans containing test cases.
- **Azure Artifacts**: Share your own NuGet package or any other packages.
- **Build pipeline**: Build code for creating application packages.
- **Release pipeline**: Deploy application packages to different release targets.

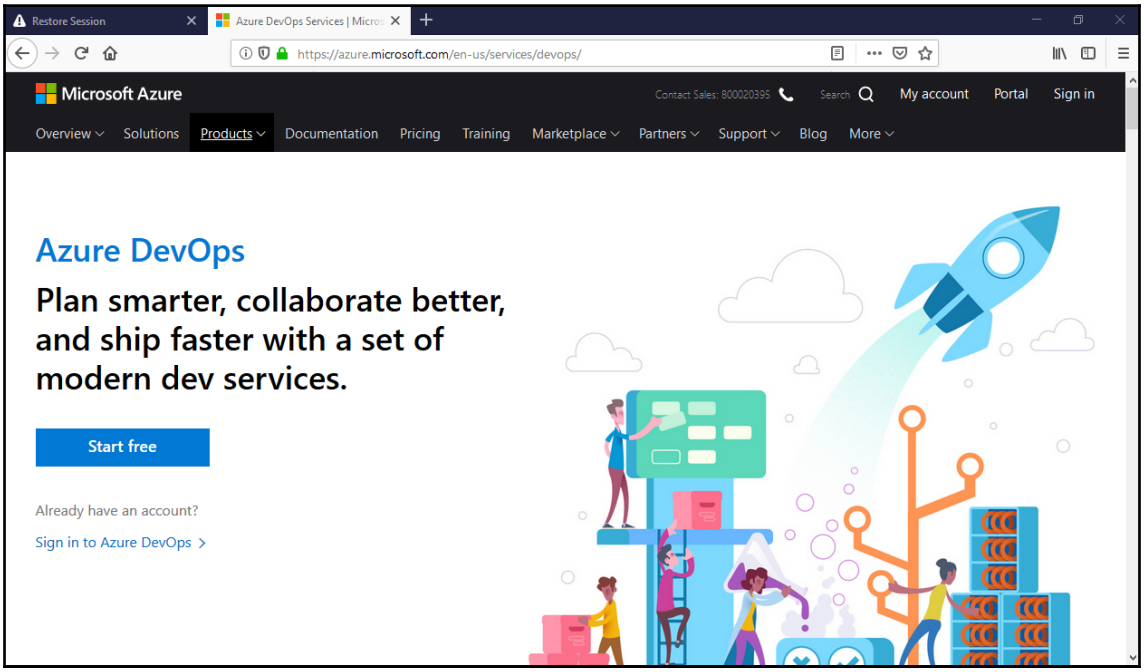
For further information on Azure DevOps and all of its features, please go to `https://azure.microsoft.com/en-us/services/devops/`.

WOW! eBook
www.wowebook.org

Creating a free Azure DevOps subscription and your first Azure DevOps project

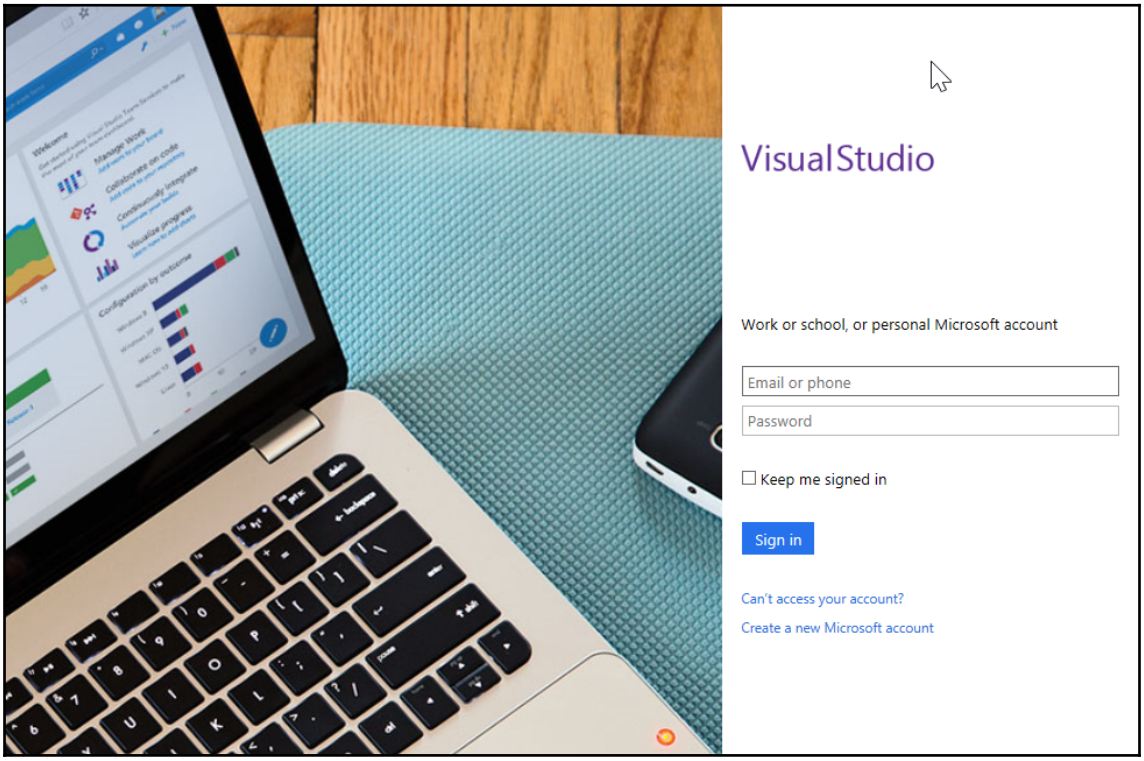
We will now explain how to create your own free Azure DevOps subscription and your first Azure DevOps project. You are going to use this information later to try out and understand the examples illustrated within this book:

1. Go to `https://azure.microsoft.com/en-us/services/devops/` and click on the **Start free** button:



WOW! eBook
www.wowebook.org

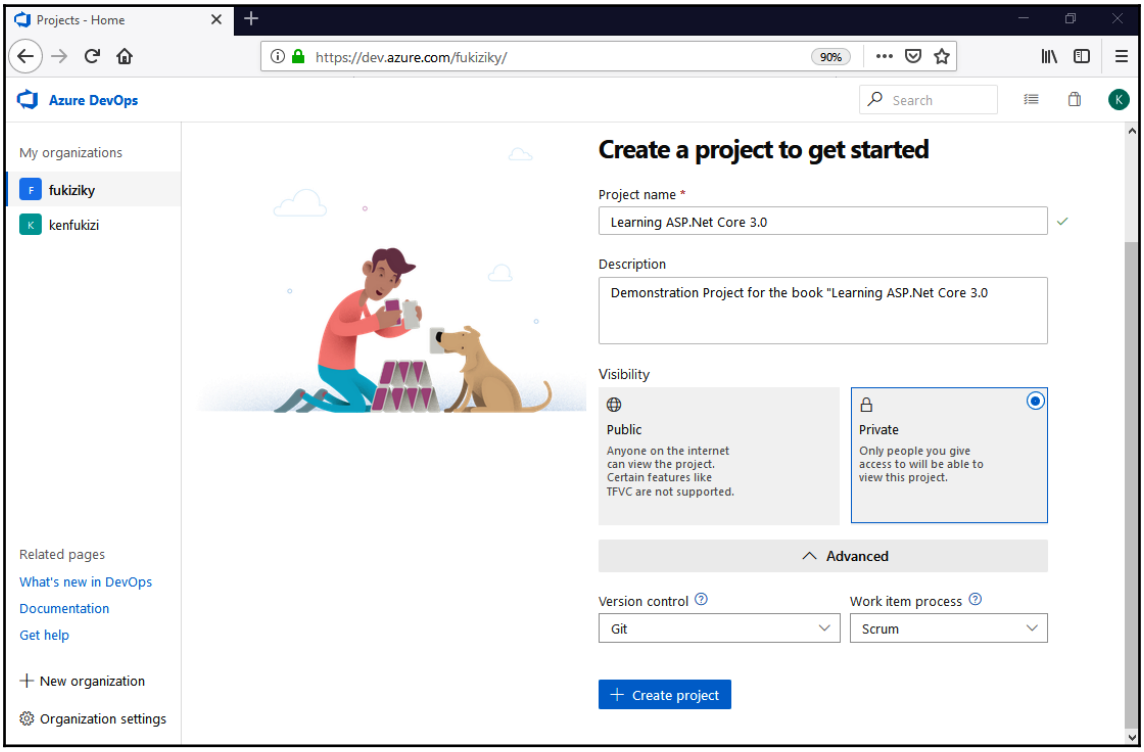
2. Log in with your work, school, or personal Microsoft account:



3. If you are connecting for the first time, enter additional information such as your name, your country, and your email address, and then click on **Continue**.

WOW! eBook
www.wowebook.org

4. Now that your account is created, let's create a new project. For our example, select **Git** as **Version control**, click on **Change Details,** and then choose **Work item process** as **Scrum**:



5. Your new project gets generated, and you are now ready to create your first work items and Git repositories, as will be shown in this chapter.

Before you do any application, it is recommended to plan for it. Azure DevOps helps you in this respect by allowing you to create and manage work items, and we will have a look at this feature in the next section.

Organizing your work via work items

Work items are used to plan, assign, track, and— more generally speaking—organize your work during a software development project. They help to better understand what needs to be done and give insights on the status of your project.

WOW! eBook
www.wowebook.org

Some common work item usages are as follows:

- Create, prioritize, and track user stories for application features.
- Create and track development tasks necessary to implement user stories.
- Create, prioritize, and track application bugs.
- Determine application quality and application release dates.
- Display the progress of user stories, tasks, and bugs in a single Kanban board.

As you have seen before, you can choose the work item process during the Azure DevOps project creation. This choice defines the standard **work item types** (**WITs**) available (agile, basic, CMMI, and scrum work item processes).

There are more than 14 WITs by default, and you can create your own custom WITs for advanced scenarios. Most of the time, you will not need to create your own custom WITs.

Possible work item process choices are as follows:

- Scrum, if your team uses the scrum methodology and if you want to track your **product backlog items** (**PBI**) on a Kanban board.
- Agile, if your team practices an agile methodology but does not want to comply with specific scrum constraints and terminologies.
- CMMI, if your team requires a more formal development task follow-up. With this, you can track requests, changes, risks, and reviews.

Here is a list of WITs, depending on the work item process:

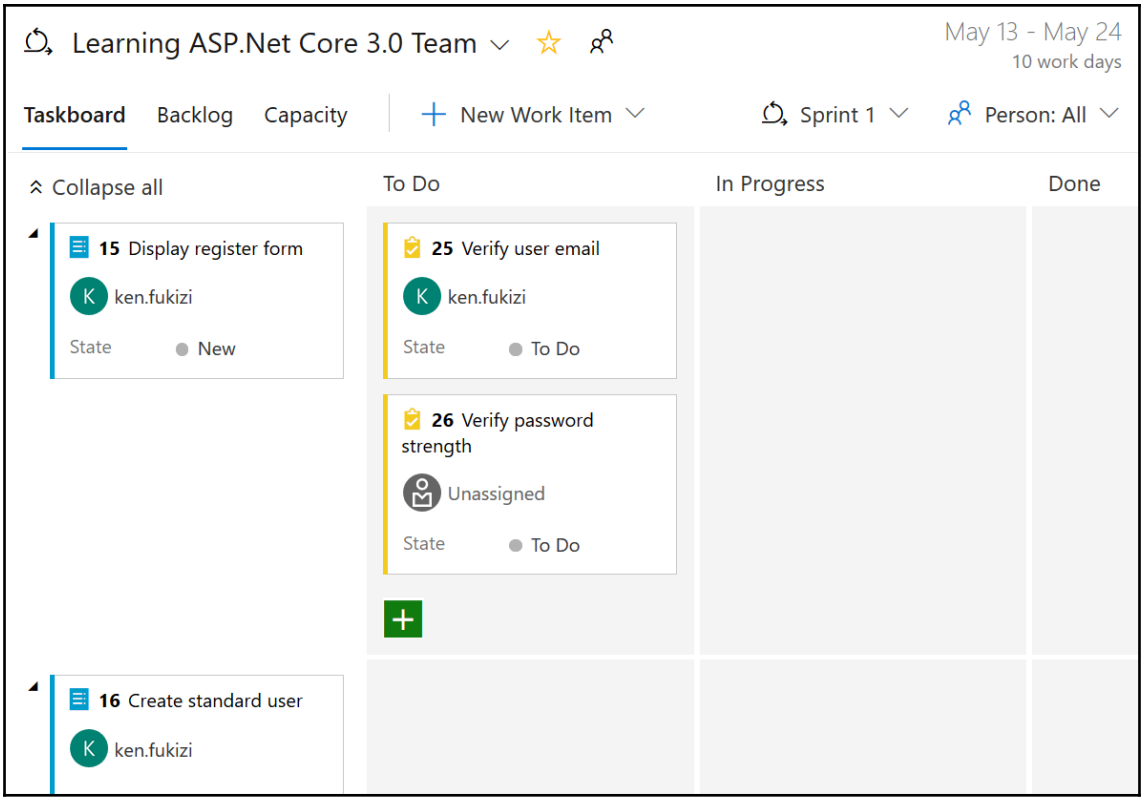
Domain	**Scrum**	**Agile**	**CMMI**
Product planning	PBI Bug	User story Bug	Requirement Change Bug
Portfolio	Epic Feature	Epic Feature	Epic Feature
Task and sprint planning	Task	Task	Task
Bug backlog management	Bug	Bug	Bug
Issue and risk management	Impediment	Issue	Issue Risk Review

WOW! eBook
www.wowebook.org

In our examples, we have chosen to use the scrum process. This methodology is one of the most commonly used in the world to manage and track work items, and if we understand scrum, it will be easier to apply the knowledge to other scenarios. Therefore, we look at scrum practical processes next, in the following section.

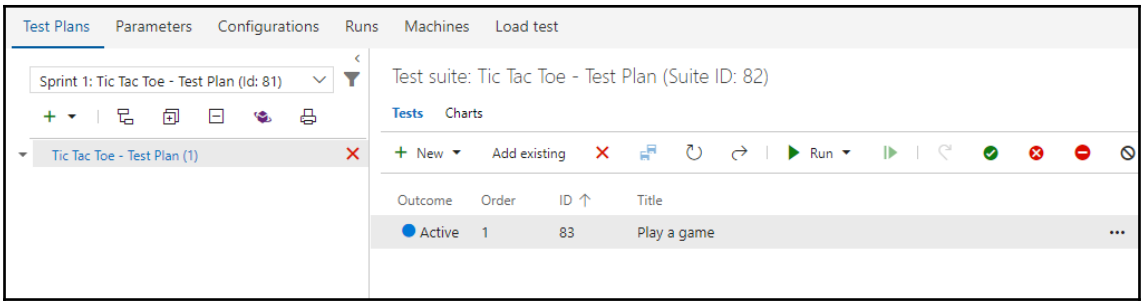
Understanding the scrum process

In the scrum process, product owners create epics, features, and product backlog items (the equivalent to user stories). During the sprint planning development, tasks are defined and linked to product backlog items. Everything is visible to the whole team via a Kanban board in the cloud:



Testers create and execute test cases by using the Azure DevOps web portal or Microsoft Test Manager. They create and assign bugs, and code defects and blocking issues can be tracked, just like the following screenshot shows:

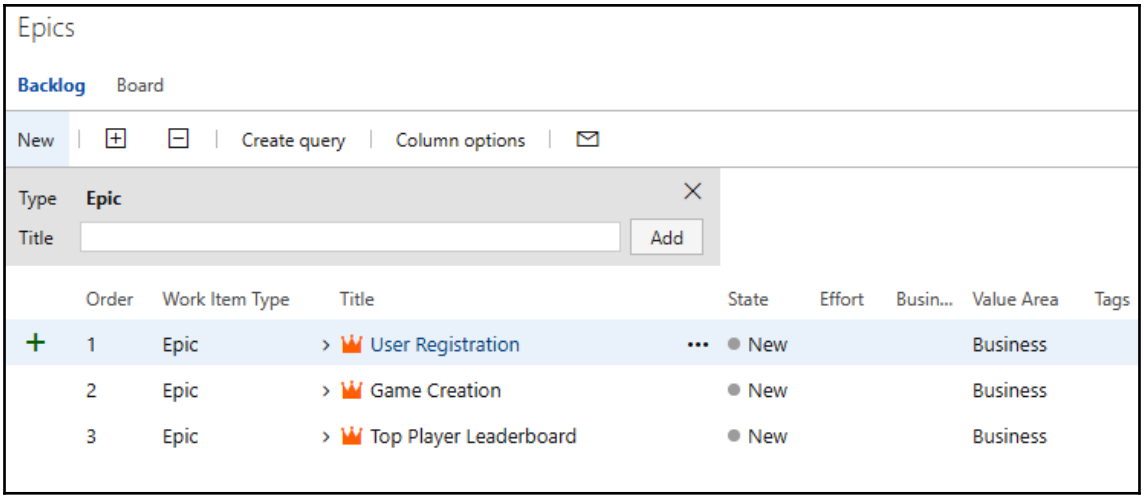
WOW! eBook
www.wowebook.org



Azure DevOps allows you to hierarchically organize your work. You can drill up, drill down, reorder, and modify parent items, as well as use filters in hierarchical views.

For even more information, go to `https://www.visualstudio.com/en-us/docs/work/backlogs/create-your-backlog`.

Let's now look at the different elements in more detail. An epic can be described as a large user story with a large amount of work. It must be broken down into features and smaller product backlog items to be able to fully understand its requirements, and then be implemented efficiently during multiple sprints:



WOW! eBook
www.wowebook.org

Features decompose epics into smaller comprehensible parts. They consist of a group of product backlog items that correspond to the detailed expected functionalities:

Features

Backlog Board

New | Create query | Column options

Type Feature

Title Add

Order	Work Item Type	Title	State	Effort	Busin...	Value Area	Tags
1	Feature	Register standard user	New			Business	
2	Feature	Register Facebook user	New			Business	
3	Feature	Register Google User	New			Business	
4	Feature	Create a new game	New			Business	
5	Feature	Login player	New			Business	
6	Feature	Invite another player	New			Business	
7	Feature	Mark space	New			Business	
8	Feature	Finish game	New			Business	
9	Feature	List top players	New			Business	

A product backlog item is a unit of work that has business value and is small enough to be completed during a single sprint. If you cannot finish it in a single sprint, then it has to be considered a feature, and must be decomposed further:

Learning ASP.Net Core 3.0 Team

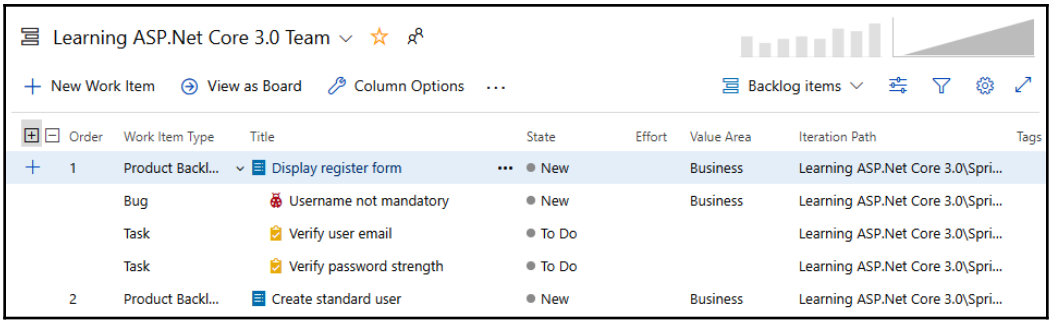
New Work Item | View as Board | Column Options | ... | Backlog items

Order	Work Item Type	Title	State	Effort	Value Area	Iteration Path	Tags
1	Product Backl...	Display register form	New		Business	Learning ASP.Net Core 3.0\Spri...	
2	Product Backl...	Create standard user	New		Business	Learning ASP.Net Core 3.0\Spri...	
3	Product Backl...	Display confirmation	New		Business	Learning ASP.Net Core 3.0\Spri...	
4	Product Backl...	Connect user with Facebook account	New		Business	Learning ASP.Net Core 3.0\Spri...	
5	Product Backl...	Register user with Facebook account	New		Business	Learning ASP.Net Core 3.0\Spri...	
6	Product Backl...	Connect user with Google account	New		Business	Learning ASP.Net Core 3.0\Spri...	
7	Product Backl...	Register user with Google account	New		Business	Learning ASP.Net Core 3.0\Spri...	
8	Product Backl...	Display form to create new game	New		Business	Learning ASP.Net Core 3.0\Spri...	
9	Product Backl...	Login user	New		Business	Learning ASP.Net Core 3.0\Spri...	
10	Product Backl...	Invite another user	New		Business	Learning ASP.Net Core 3.0\Spri...	

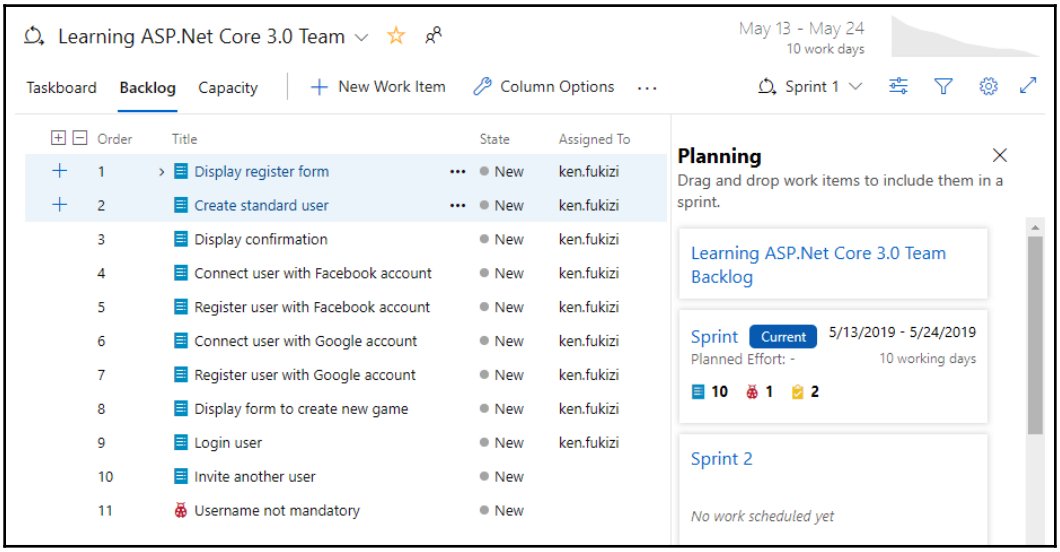
WOW! eBook
www.wowebook.org

Tasks describe the development or testing work necessary for implementing the expected product backlog item functionalities during the sprint. They are linked to product backlog items for trackability and are able to automatically calculate project advancement.

During a sprint, there are times when a finished task does not entirely do what it was meant to do in the correct way, or it may cause other parts of a system to behave incorrectly. These are called **bugs** and contain issues that have been raised by testers and/or system users, within the duration of a sprint, which is typically organized in two-week cycles. Bugs may be assigned to be resolved during a sprint, and they are linked to their corresponding product backlog items:

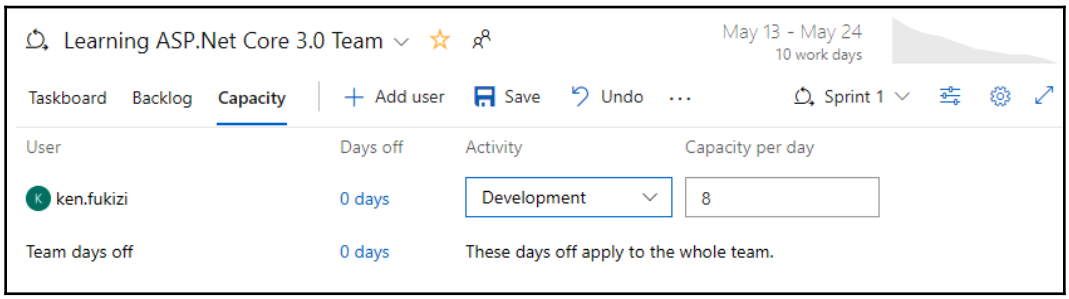


After defining epics, features, and product backlog items, you can do your sprint planning and decide what needs to be done in which iteration. Additionally, the Kanban board provides a great visual representation, for better understanding:



WOW! eBook
www.wowebook.org

The working capacity for each team member can be defined for each sprint, and a work details report allows you to follow their work achievements in real time:



Furthermore, each work item has a state that changes over time. The state allows you to track work achievements and filter work items, for better understanding and to detect issues.

The following table shows the various default work item states, depending on the work item process:

	Scrum	**Agile**	**CMMI**
Work Item States	New Approved Committed Done Removed	New Active Resolved Closed Removed	Proposed Active Resolved Closed

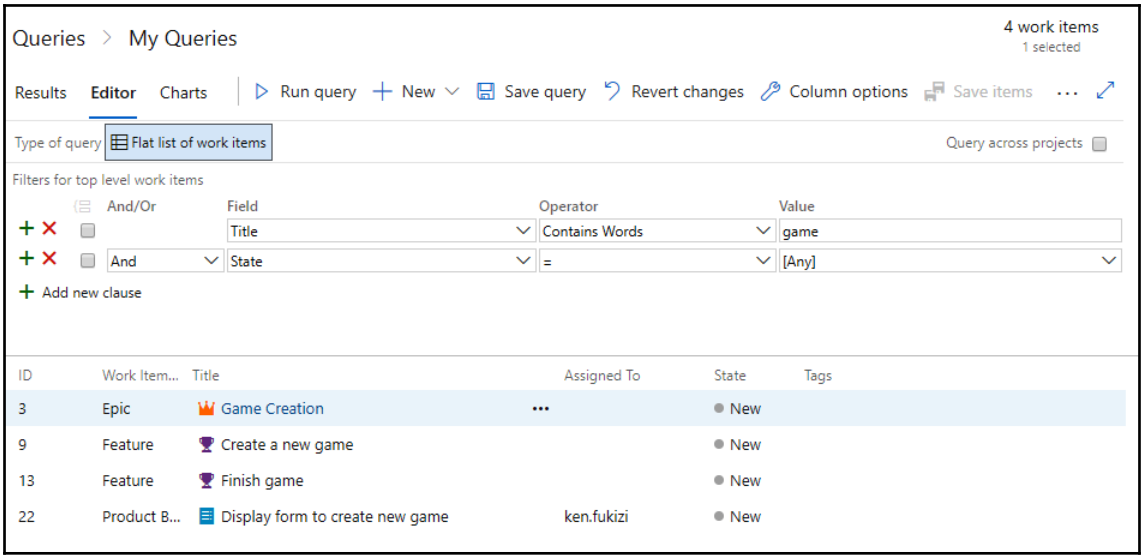
Please note that you do not have to follow each status, as defined for scrum, agile, or CMMI. You can customize and add in different statuses as you see fit in your specific organization. For example, there are other enterprises that decide to add in custom statuses to complement the existing steps, as follows:

- **Committed-Developed** (development is done, ready for QA)
- **Committed-Tested** (QA is finished, ready for product owner demo and sign-off).

You can query for work items, create graphs, and publish them to your Azure DevOps project home page. This is a very useful feature if you need to retrieve specific work items or need to get a holistic view of your project.

WOW! eBook
www.wowebook.org

Now, let's look at the following screenshot:



The preceding screenshot shows a query for work items whose title contains the word **game**, and respective results are shown in the same window on a lower pane.

Using Git as a VCS

Over the last few years, Git has had considerable success and is now the preferred distributed VCS among the developer community.

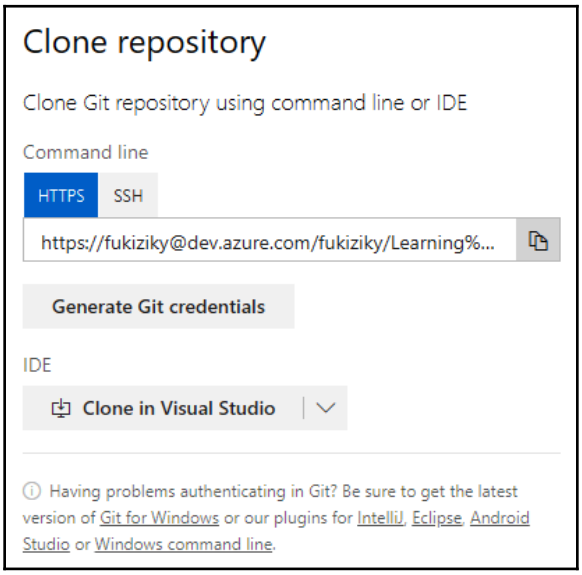
There is great integration between Azure DevOps and Git, and you have some powerful and productive features at your disposal (`https://www.visualstudio.com/en-us/docs/work/backlogs/connect-work-items-to-git-dev-ops`), including the following:

- Git branches can be created from within your backlog or Kanban board.
- Git feature branches can easily be created for multiple work items, directly from the Azure DevOps website.
- Pull requests and commits are automatically linked to corresponding work items.
- A build **Summary** page shows work items, which are linked to a commit, as associated work items.

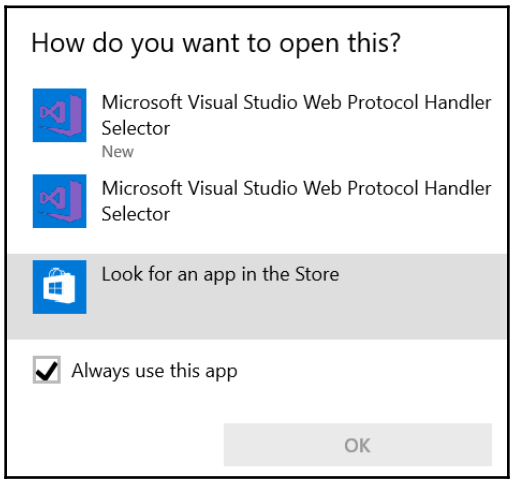
WOW! eBook
www.wowebook.org

Let's see how to create a new Git repository, clone it locally, use it within Visual Studio 2019, and create your first commit:

1. In your Azure DevOps project, click in the left-hand menu on **Repos**, and then click on the **Clone in Visual Studio** button:

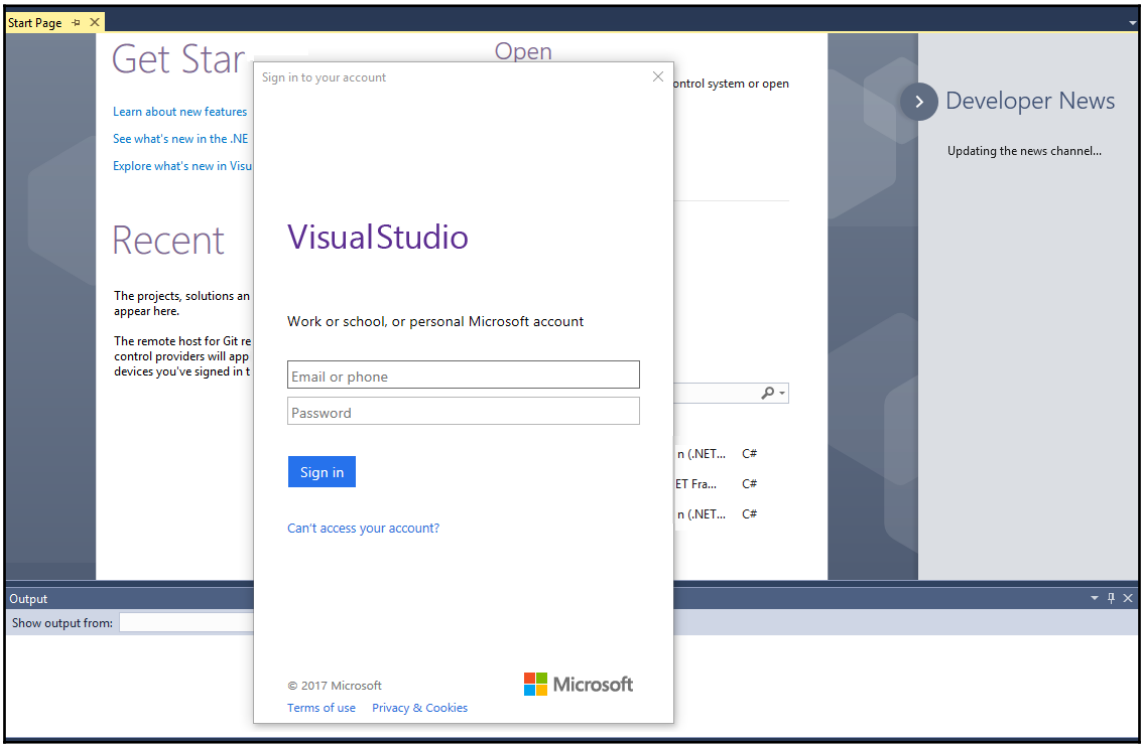


2. A new window will be displayed; select **Microsoft Visual Studio Web Protocol Handler Selector**:

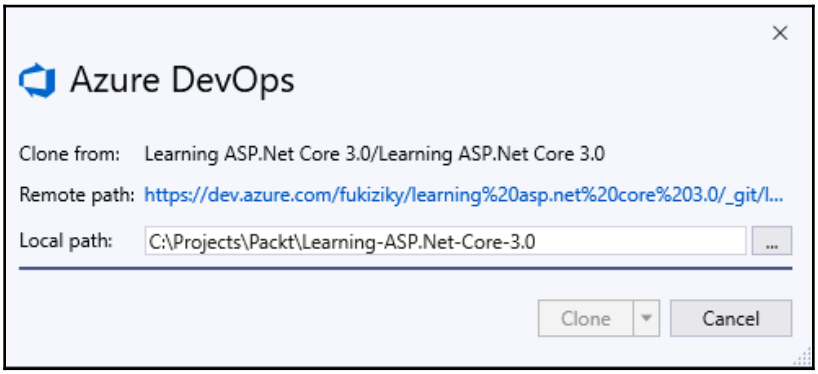


WOW! eBook
www.wowebook.org

3. Visual Studio 2019 is started automatically, and you can authenticate with your work, school, or personal Microsoft account:

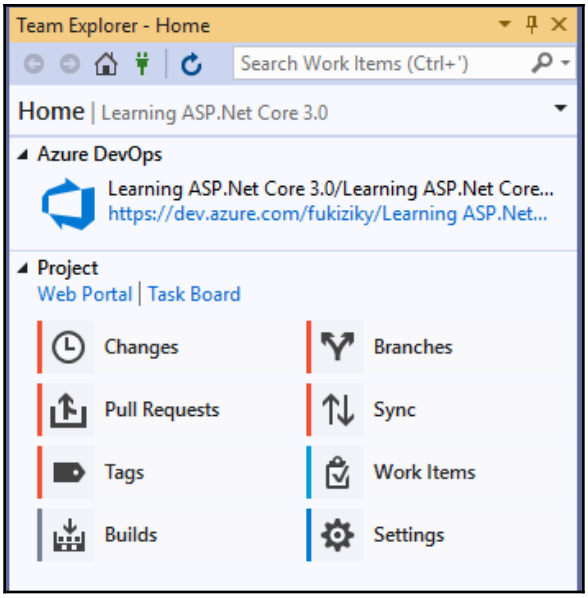


4. Choose the destination folder for your local Git repository, and click on the **Clone** button to start the download:

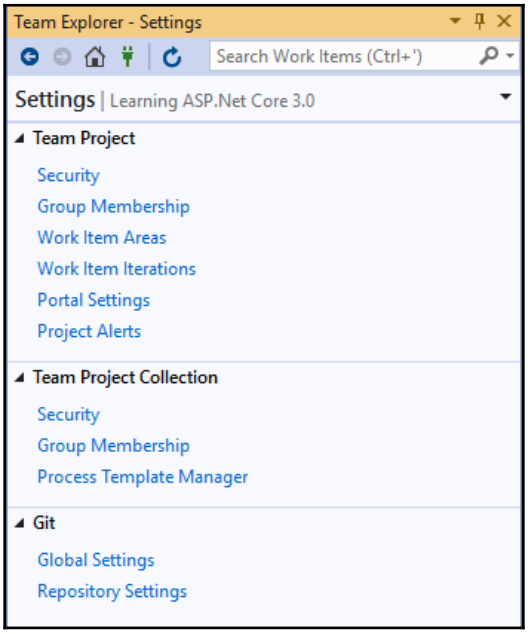


WOW! eBook
www.wowebook.org

5. Go to **Team Explorer - Home** and click on **Settings**:

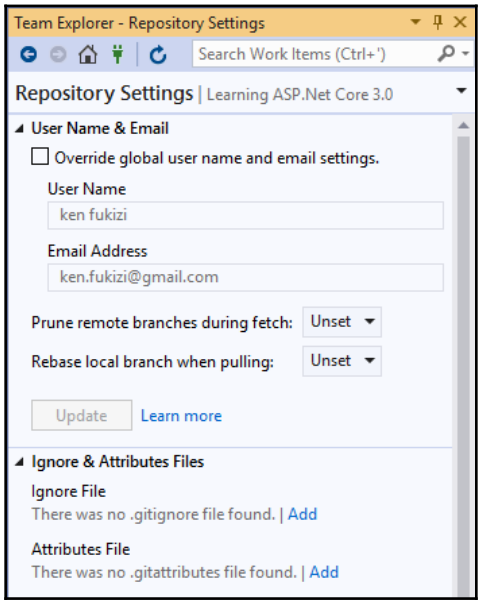


6. In **Team Explorer - Settings**, click on **Repository Settings**:

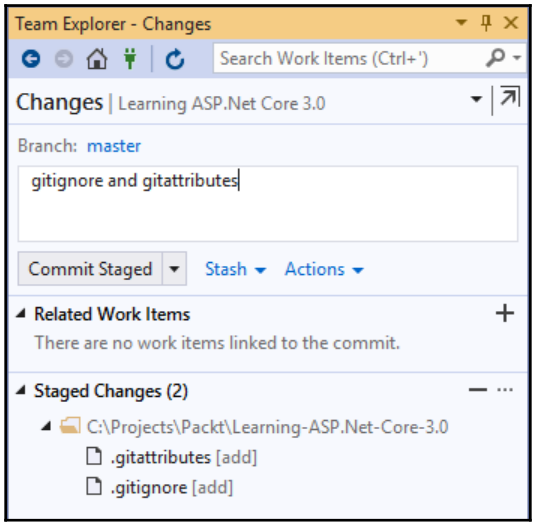


WOW! eBook
www.wowebook.org

7. In the **Ignore & Attributes Files** section, click on **Add** for the ignore file and the attributes file:

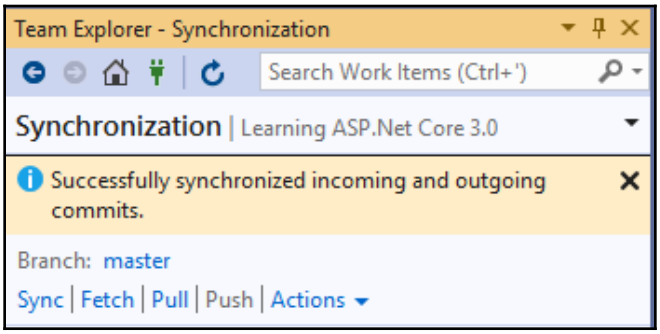


8. Return to **Team Explorer - Home**, and this time click on **Changes**, enter a comment for your first commit, and click on the **Commit Staged** button:

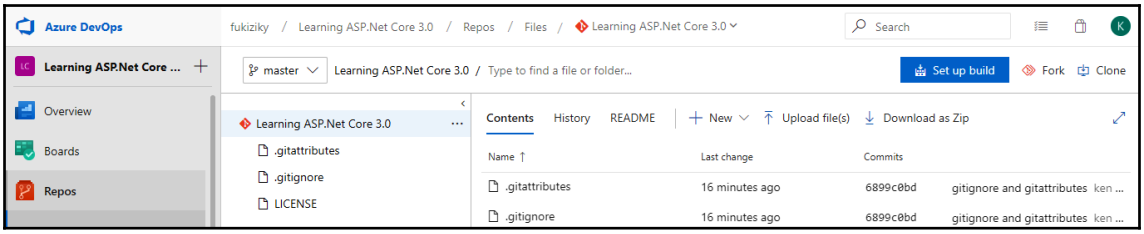


WOW! eBook
www.wowebook.org

9. Your first commit was created locally when you clicked on the **Commit Staged** button; click on the **Sync** link to push it to the server:



10. Go to the Azure DevOps website and click on **Code** in the upper menu; you can see that your created files have been uploaded:



That's it! You have created and initialized your Git repository. It's as easy as that! From here, you have multiple paths you can follow. For instance, leaving everything in the same branch is not really a very good idea, especially when you have to maintain multiple versions of your application.

For guidance on different branching strategies, see `https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops`.

WOW! eBook
www.wowebook.org

Using feature branches

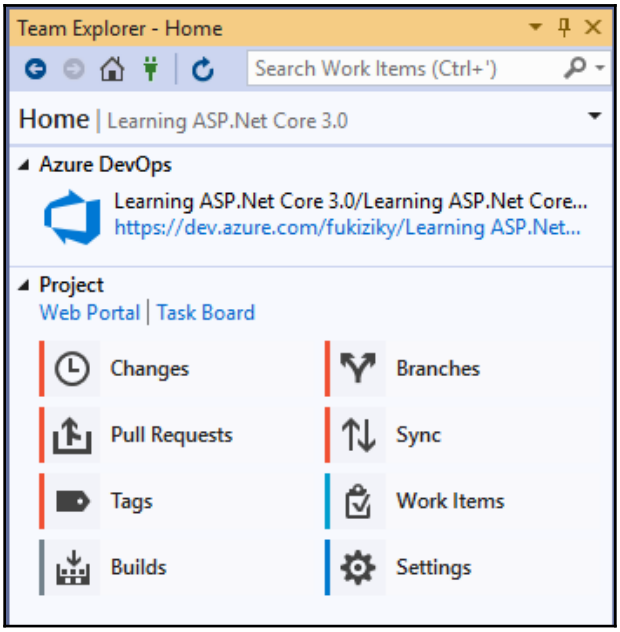
The philosophy behind feature branches is that the first thing you have to do each time you begin working on a new Azure DevOps feature (or even, Azure DevOps product backlog item) is to create a new so-called feature branch.

You then work on this branch in complete isolation until you are ready to push your tested and validated modifications to your master branch (or, in more sophisticated environments, your development branch). Until it is pushed, it will not interfere with your other features, neither will it cause bugs or lower the overall quality.

If a project deadline approaches and you have not finished all of the planned features in time, you do not need to stress anymore! Why? Because you can integrate only the features that are ready for release. You will have a product with fewer features, but you can be confident that those features are going to work as expected, without any risks.

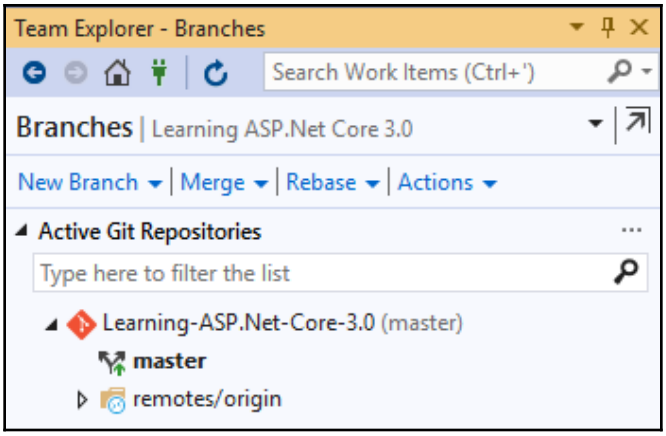
Let's look at how to create a feature branch, using Visual Studio 2019 and Git:

1. Open Visual Studio 2019, go to the **Team Explorer - Home** tab, and click on the **Branches** button:

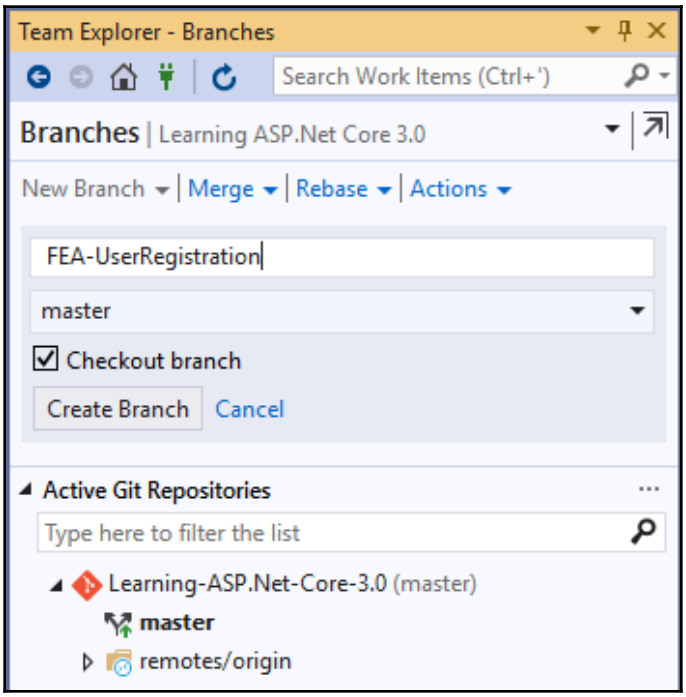


WOW! eBook
www.wowebook.org

2. In **Team Explorer - Branches**, click on the **New Branch** link:



3. Enter a new feature branch name (use the `FEA-` prefix), and then click on the **Create Branch** button:



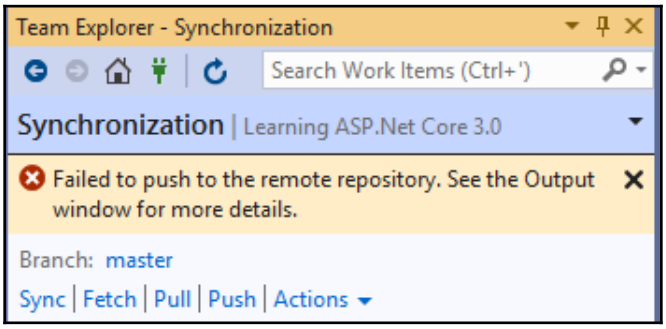
WOW! eBook
www.wowebook.org

It must be noted that we're using the `FEA-` prefix just as good practice, to enable other members of the team to identify that this is a feature branch. It is not mandatory to put the `FEA-` prefix. Different teams agree on different naming conventions for branches.

Merging changes and resolving conflicts

Sometimes, team members work on the same files at the same time, leading to conflicts. Let's see how to merge changes and resolve conflicts in such a scenario:

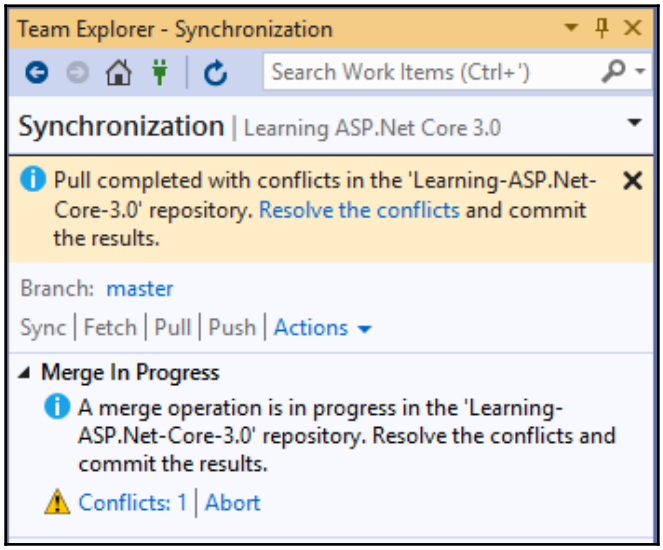
1. Create a text file called `HelloWorld.txt` and add it to your local repository. Push the file to the server, and update the file both on the server and in your local repository.
2. If you try to push a `HelloWorld.txt` file that has been modified both locally and in the remote repository, you get an error message, and the push fails:



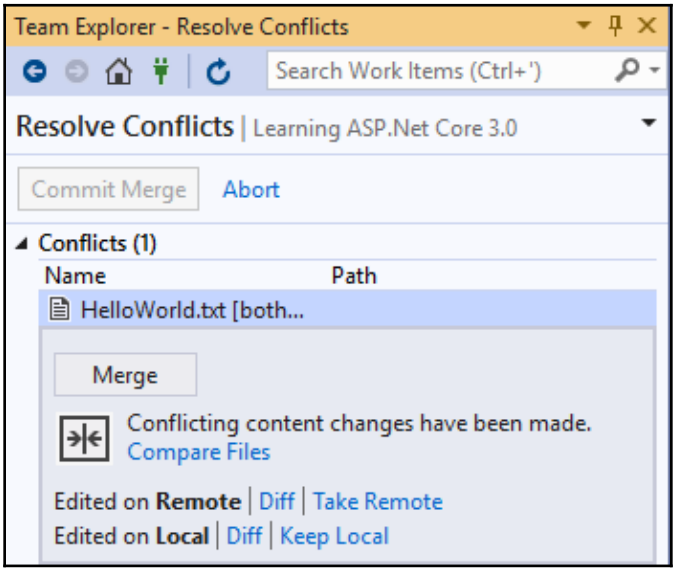
3. When looking in the output window, you get additional information on the possible reason why your push failed, as follows: *Error: Hint: Updates were rejected because the remote contains work that you do not have locally*. This is usually caused by another repository pushing to the same reference. You may want to first integrate the remote changes (for example, `git pull`) before pushing again.

WOW! eBook
www.wowebook.org

4. Click on the **Pull** link and you will get the remote changes, which will result in a conflict between your local copy and the remote one. Click on either the **Resolve the conflicts** or the **Conflicts** link:

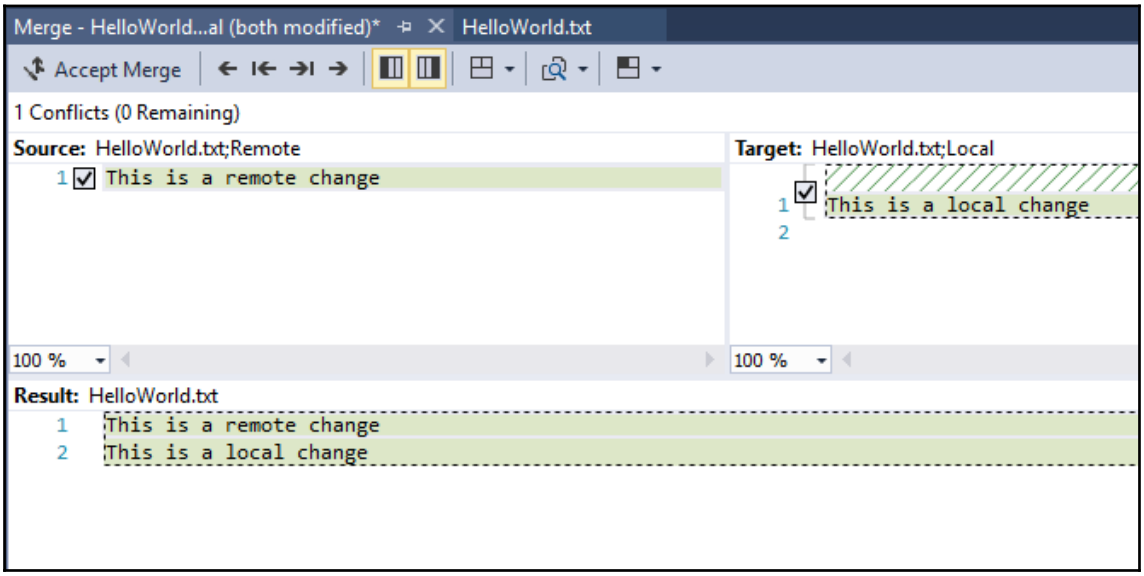


5. You will now see a list of conflicting files. Click on the conflict you want to resolve, and then click on the **Merge** button:

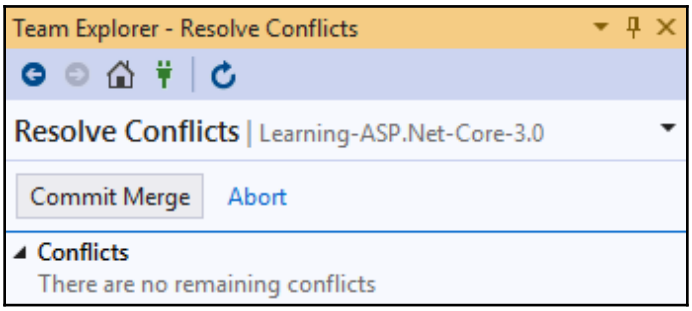


WOW! eBook
www.wowebook.org

6. You will see the conflicting modifications. Choose which ones you want to keep (the left one, the right one, or both), and click on the **Accept Merge** button:

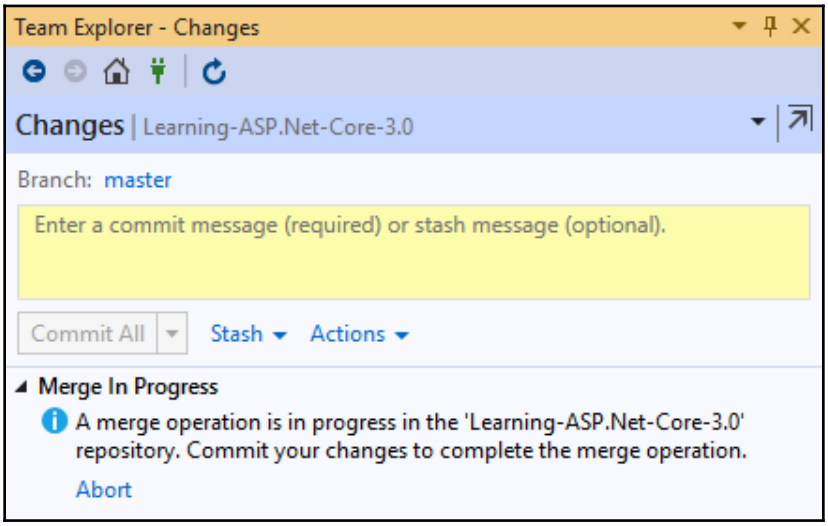


7. Back in the **Team Explorer - Resolve Conflicts** window, click on the **Commit Merge** button:

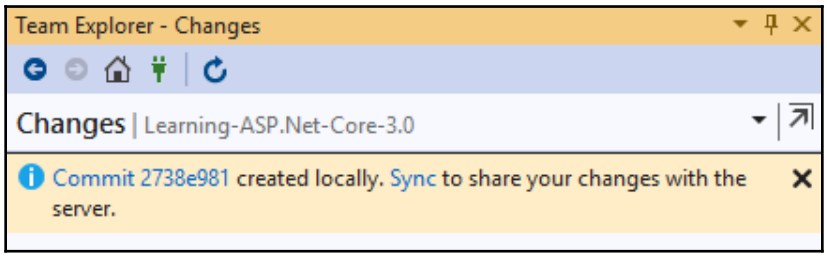


WOW! eBook
www.wowebook.org

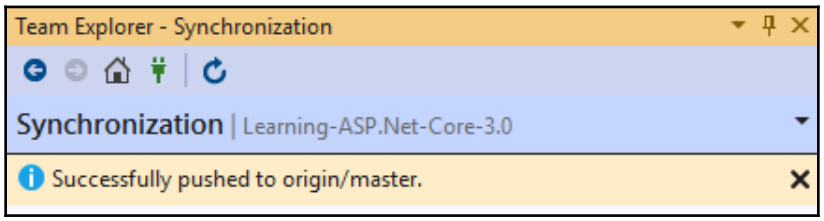
8. Enter a comment, and click on the **Commit All** button to finalize and commit the merge locally:



9. After the commit has been created locally, click on the **Sync** link, and then on the **Push** link:



10. You should now see that the changes have been uploaded to the remote repository:



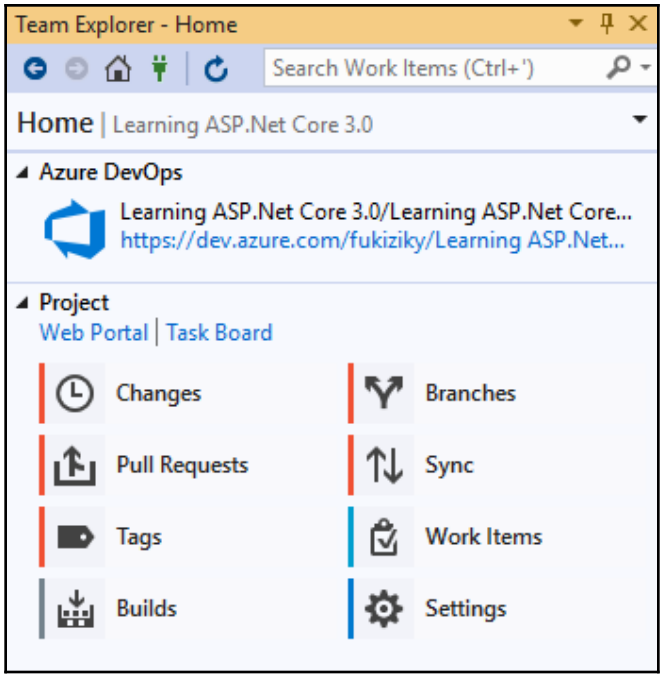
WOW! eBook
www.wowebook.org

In this section, we have seen the basic usage of Git, using mainly the Visual Studio development environment, but it must also be noted that you can also use the command line to effect the same commands. There are other software applications—such as GitHub Desktop, Git Extensions, and many more—that are designed to help you interact with your repositories, as an abstraction, but they all use the same `git` commands as underlying instructions to the VCS, and they all use similar terminology for most commands.

Creating an Azure DevOps build pipeline

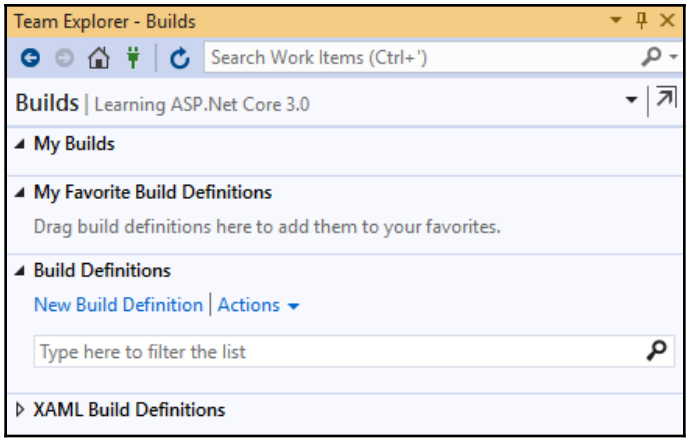
After having planned and organized your work and created your Git repository, you should now configure an Azure DevOps build pipeline, which will allow you to do CI for your application:

1. Open Visual Studio 2019, go to the **Team Explorer - Home** tab, and then click on the **Builds** button:

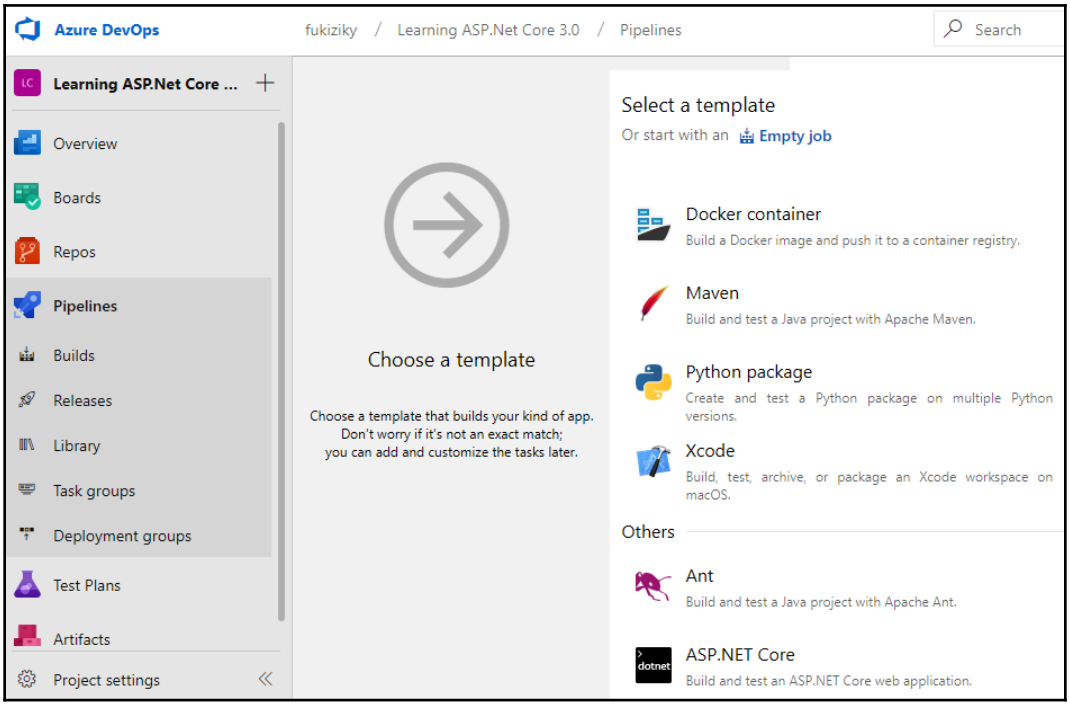


WOW! eBook
www.wowebook.org

2. Next, click on the **New Build Definition** link:

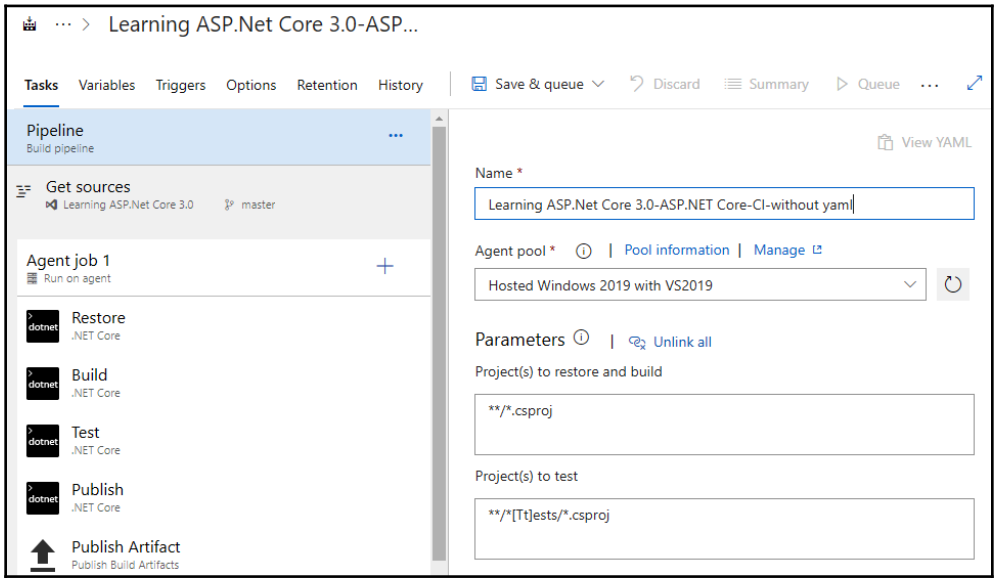


3. The Azure DevOps website is opened, and when you click the new pipeline button and then select your source as Azure Repos, you are presented with a choice of build definition templates. Select the **ASP.NET Core** template:

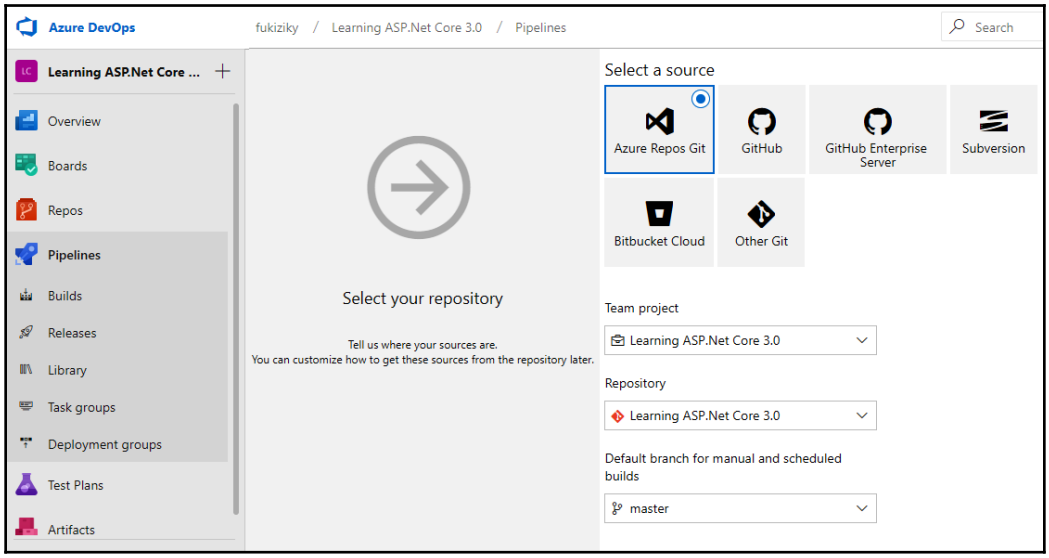


WOW! eBook
www.wowebook.org

4. In the new build definition, enter a name, and select your default agent pool. We recommend using the option **Hosted Windows 2019 with VS2019**:

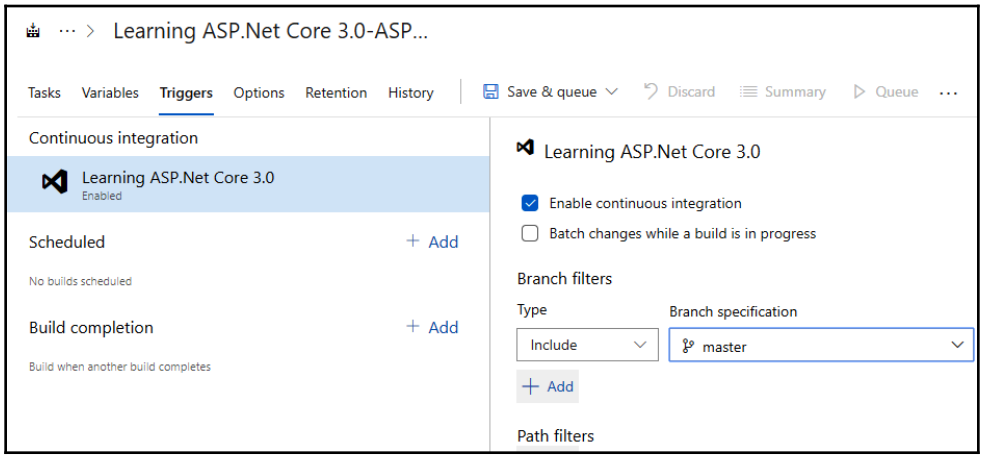


5. To choose a source repository, click on **Get sources**. For our example, we use the default values (this project, branch: **master**):

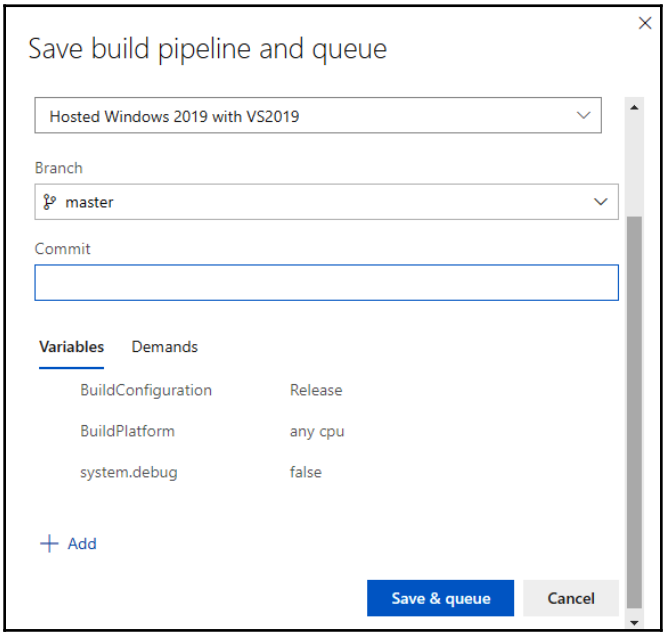


WOW! eBook
www.wowebook.org

6. To enable CI, click on **Triggers** in the build definition menu, and then tick the **Enable continuous integration** checkbox:



7. After verifying that the Git repository and master branch have been selected correctly, click on the **Save** or **Save & queue** button. The configuration has been finished, and a build will automatically be triggered each time code is committed to the repository:



WOW! eBook
www.wowebook.org

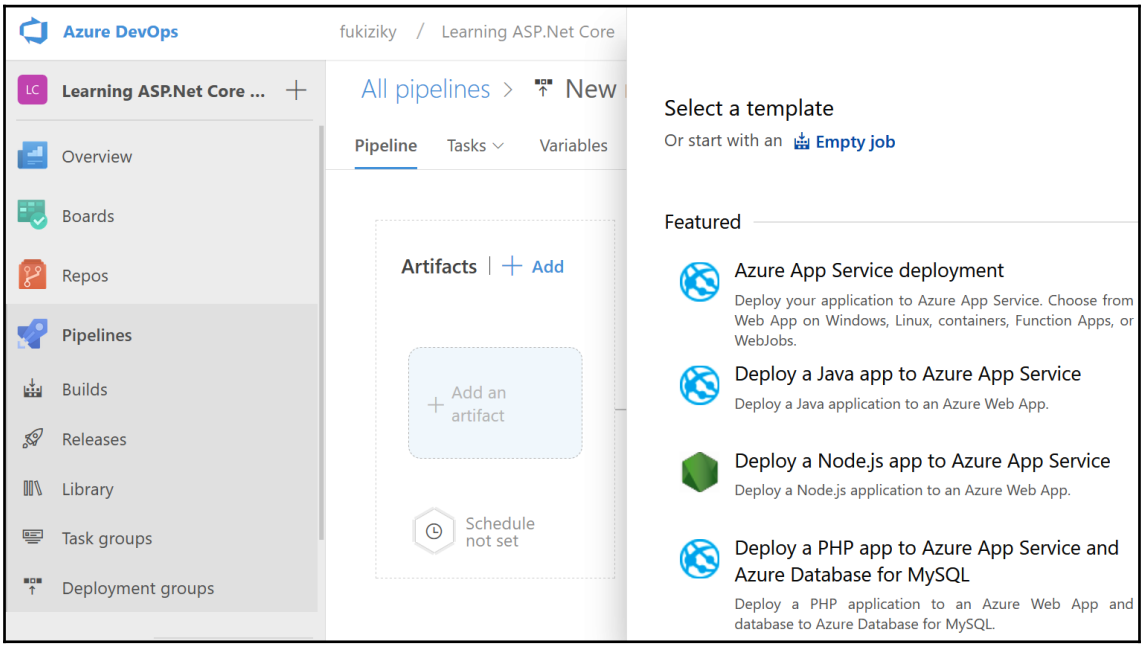
Creating a build pipeline is as simple as that. After building, it's only natural that we want to release our code, so we will look at how we can create a release pipeline next, in the following section.

Creating an Azure DevOps release pipeline

Alongside your application getting integrated continuously, you have also seen some great benefits, such as detecting and fixing bugs, and other issues, much faster. Let's not stop there; improving your development process even further is much easier than you think!

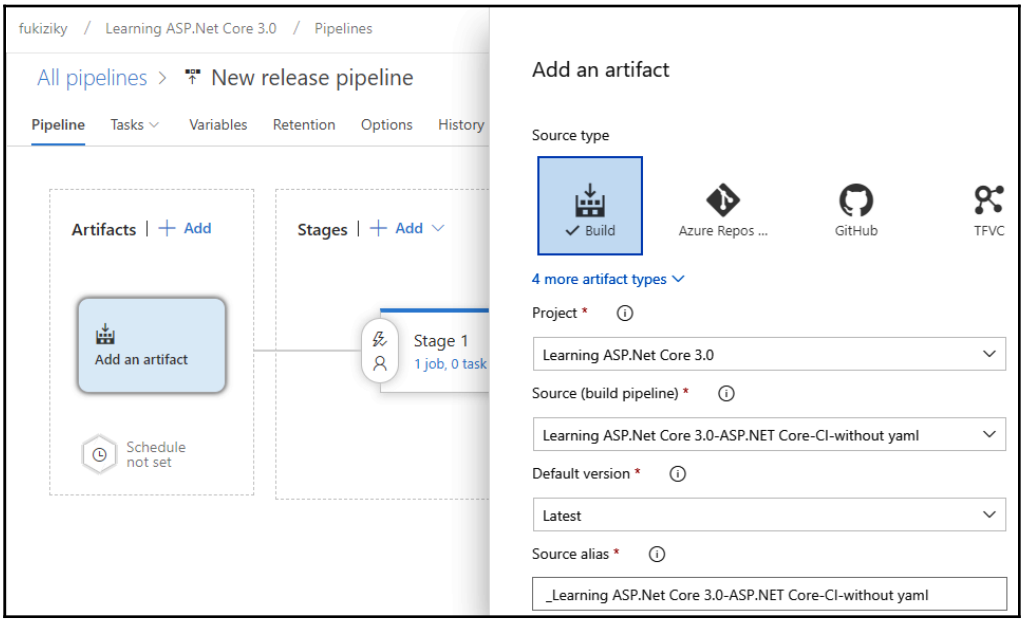
We will now see how to adopt the CD of your application by creating an Azure DevOps release pipeline:

1. Open the Azure DevOps website, click on **Pipelines** in the menu, click on **Releases**, then on the **New definition** button, and then select the **Empty job** definition template:



WOW! eBook
www.wowebook.org

2. You can now select the **Project** and the **Source (build pipeline)**, enable the CD, and then click on the **Create** button:



3. The release definition gets created, and you can see it in the list.

The sample release definition shown does not really do very much for now. We will see a much more advanced version later that deploys to Azure, in the corresponding Azure chapters.

Summary

In this chapter, we have learned about CI, CD, and build and release pipelines, including what the benefits are and how to implement them using Azure DevOps.

We have created a new Azure DevOps subscription and have initialized a new project. We then explored some of the basic concepts, such as work items and Git for source control. Finally, we illustrated how to configure an Azure DevOps build pipeline, as well as an Azure DevOps release pipeline, via a practical example.

In the next two chapters, we will explain the basic concepts of ASP.NET Core 3, including the start up class, using middleware, routing, error handling, and much more.

WOW! eBook
www.wowebook.org

2
Section 2: A Practical Demonstration of ASP.NET Core 3

In this section, you will learn how to develop a real-world ASP.NET Core 3 application, from the basics all the way to having a fully functional application. Toward the end of this section, we'll piece together all the basic concepts we introduced earlier into a usable application.

This section comprises the following chapters:

- `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*
- `Chapter 5`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 2*
- `Chapter 6`, *Introducing Razor Components and SignalR*
- `Chapter 7`, *Creating ASP.NET Core MVC Applications*
- `Chapter 8`, *Creating Web API Applications*

WOW! eBook
www.wowebook.org

4
Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1

In the last three chapters, you have seen what ASP.NET Core 3 is about from a global point of view, as well as how to set up your development environment, including Visual Studio 2019 (or Visual Studio Code). We have also seen how to set up **CI** (short for **continuous integration**) and **CD** (short for **continuous delivery**) pipelines in Azure DevOps with a Git repository.

This is all really interesting, but very theoretical. Now it is time to do something practical, time to get right to it, time to build something by yourself!

In this chapter, we are going to build an application to showcase the basic concepts of the ASP.NET Core 3 Framework. In the following chapters, we will constantly be improving this application, while using and illustrating the various features of ASP.NET Core 3 and the technologies surrounding it.

Having gone through the chapter's content, you will have acquired skills in working with ASP.NET Core 3 start up classes, targeting different .NET Frameworks, working with middleware, and performing error handling in ASP.NET Core 3.

In this chapter, we will cover the following topics:

- The `Startup` and `Program` classes
- Creating pages and services
- Using **Node Package Manager** (**NPM**) and layout pages
- Applying dependency injection
- Using the built-in middleware
- Creating your own middleware

WOW! eBook
www.wowebook.org

- Working with static files
- Using routing, URL redirection, and URL rewriting
- Error handling and model validation

Preview of the Tic-Tac-Toe demo application

Let's do something fun! We will build a Tic-Tac-Toe game, also known as **noughts and crosses**, or Xs and Os.

In this game that we will build, players will choose who takes the Xs and who takes the Os. Then, they will be taking turns to mark spaces in a 3×3 grid, one mark per turn. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game, as shown:

x		0
0	**x**	0
x	0	**x**

In the preceding diagram, the player with Xs will have won the game because there are three crosses in a diagonal row, from the top-left corner to the bottom-right corner.

Players will have to enter their emails and names for registration to create an account before being able to start a game. They will receive a game score after each match, which is going to be added to their total score.

We will have a leaderboard that is aimed at providing information on player rankings and top scores, for a series of games played.

When creating a game, a player will have to send an invitation to another player, and then a specific waiting page will be displayed for them until another player has responded.

Following receipt of the invitation email, the other player can then confirm the request and join the game. When the two players are online, the game starts.

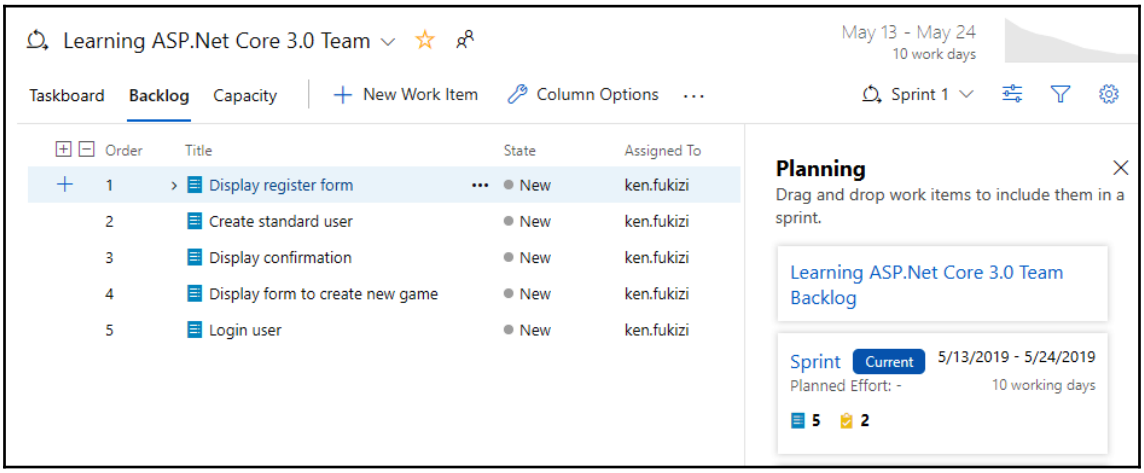
That is a preview of what we are going to build our demo application around, but talk is cheap. Let's get started building it up from the next section.

WOW! eBook
www.wowebook.org

Building the Tic-Tac-Toe game

As explained in the last chapter, we can use Azure DevOps and its work items to organize and schedule the implementation of our Tic-Tac-Toe game application. For that, we have to create epics, features, and product backlog items, and then do sprint planning to prioritize and decide what has to be implemented first.

As you can see in the following screenshot, we have decided to work on five product backlog items in the first sprint and have added them to the sprint backlog:



Do you remember what needs to be done next, before implementing any of the new features? You don't remember? Perhaps feature branches ring a bell?

In the previous chapter, we showed the best practices for creating development branches, which are isolated and easier to maintain and release. They consist of creating a feature branch in the Git repository for every new feature that you want to add to your application.

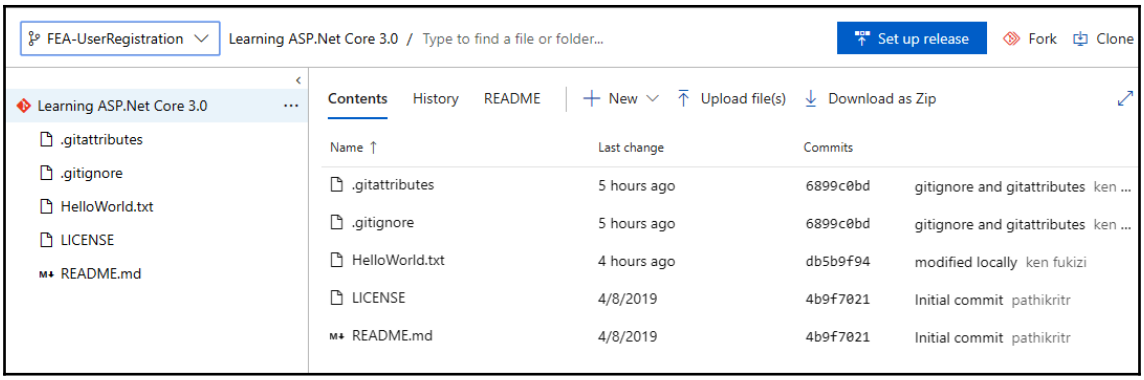
Hence, every developer can work on their specific features within their specific feature branch, until they have decided that it is ready to be released.

At the end, all of the features ready for release are merged into a development (either release or master) branch. Then, integration tests are carried out and, if everything is working as expected, a new application version is delivered.

WOW! eBook
www.wowebook.org

The feature we have chosen to work on first is user registration, so the first thing we have to do is to create a feature branch called `FEA-UserRegistration`. If you do not know how to do that, you can go to `Chapter 3`, *Continuous Integration Pipeline in Azure DevOps*, and get a full step-by-step procedure with thorough explanations.

After you have created your feature branch for user registrations in Azure DevOps, it will look as shown in the following screenshot:



At this point, this is still an empty project with only a history of previous commits, and a `HelloWorld` text file that we used to demonstrate how to resolve conflicts.

We will be creating and building an ASP.NET Core 3 solution together on a step-by-step basis, starting from the following sections.

Conceiving and implementing your first Tic-Tac-Toe feature

Before we can implement the user registration feature, we have to understand it and decide how everything should work. We have to define **user stories** and **workflows**. For that, we need to analyze the Tic-Tac-Toe game description mentioned previously in more detail, in the preview section.

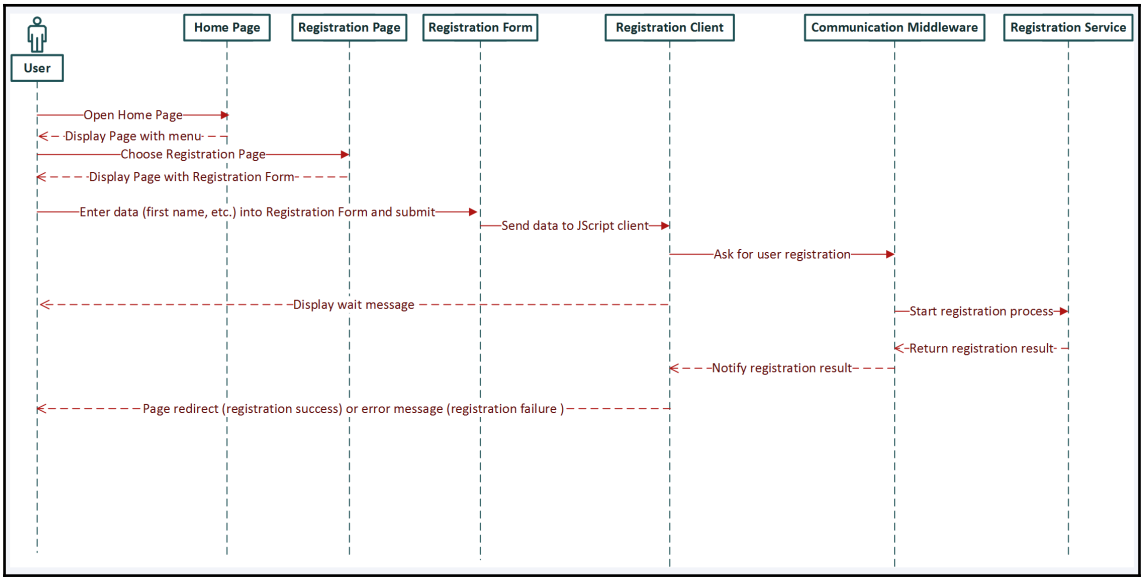
As explained previously, a user can only create and join games if they have a user account. To create this account, the user has to enter their first name, last name, email address, and a new password. The system will then verify whether the email address entered is already registered. A given email address can only be registered once. If the email address is new, the user account will be generated; if the email address is known, an error will be displayed.

WOW! eBook
www.wowebook.org

Let's look at the user registration process and the different components that have to interact in order to implement it:

1. There will be a home page with a link for user registration, where a new user must click on **Register** in order to create their player account. Clicking on the user registration link redirects the user to a dedicated registration page.
2. The registration page will contain a registration form, where the user must enter their personal information and then confirm it.
3. A JavaScript client will validate the form, submit and send the data to a communication middleware, and then await an outcome.
4. The communication middleware will receive the request and route it to a registration service.
5. The registration service will receive the request, verify the integrity of the data, check whether the email has already been used for registration, and either register the user or return an error message.
6. The communication middleware will receive the result and route it to the waiting JavaScript client.
7. The JavaScript client will then redirect the user to start playing games if the result is a success, and it will display an error message if the result is a failure.

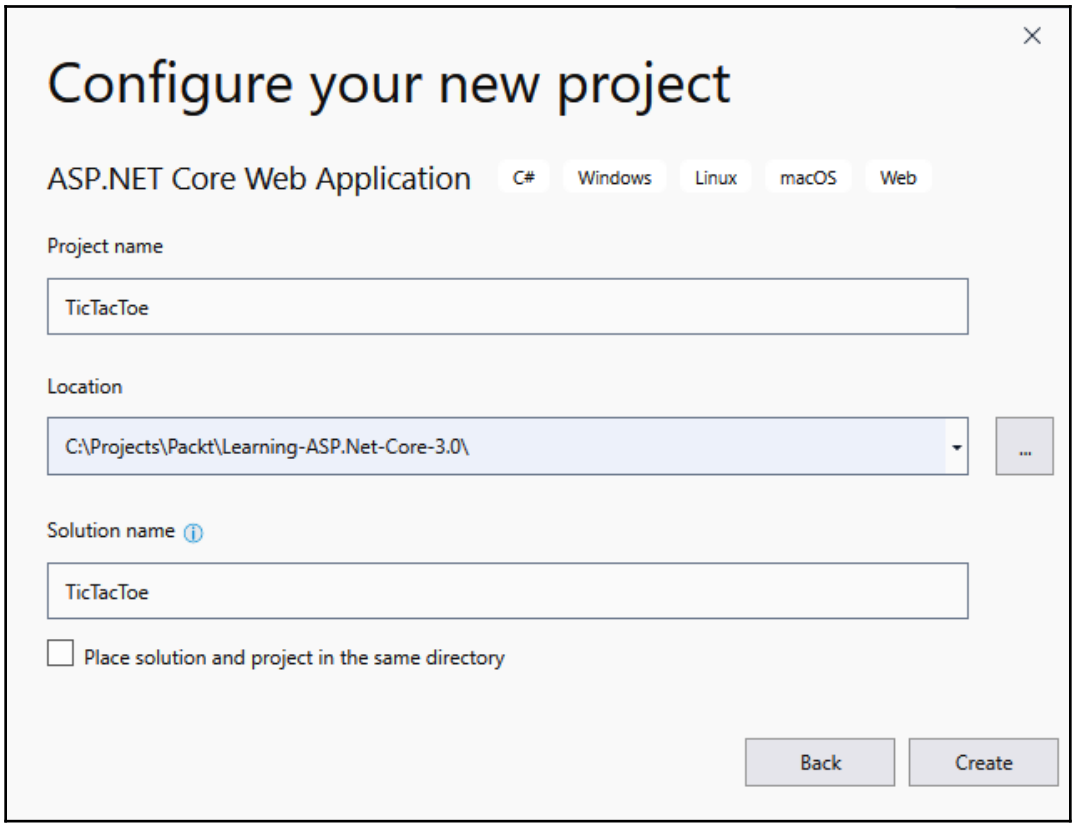
The following **sequence diagram** shows the user registration process. It is often much easier and quicker to comprehend a process with a more visual representation:



WOW! eBook
www.wowebook.org

To get started, we need to create a new empty ASP.NET Core 3 web application, which will be used for adding various components and packages in this chapter and during the remainder of the book. We will then add new concepts and functionalities progressively, which will allow you to really understand what is going on and how everything works:

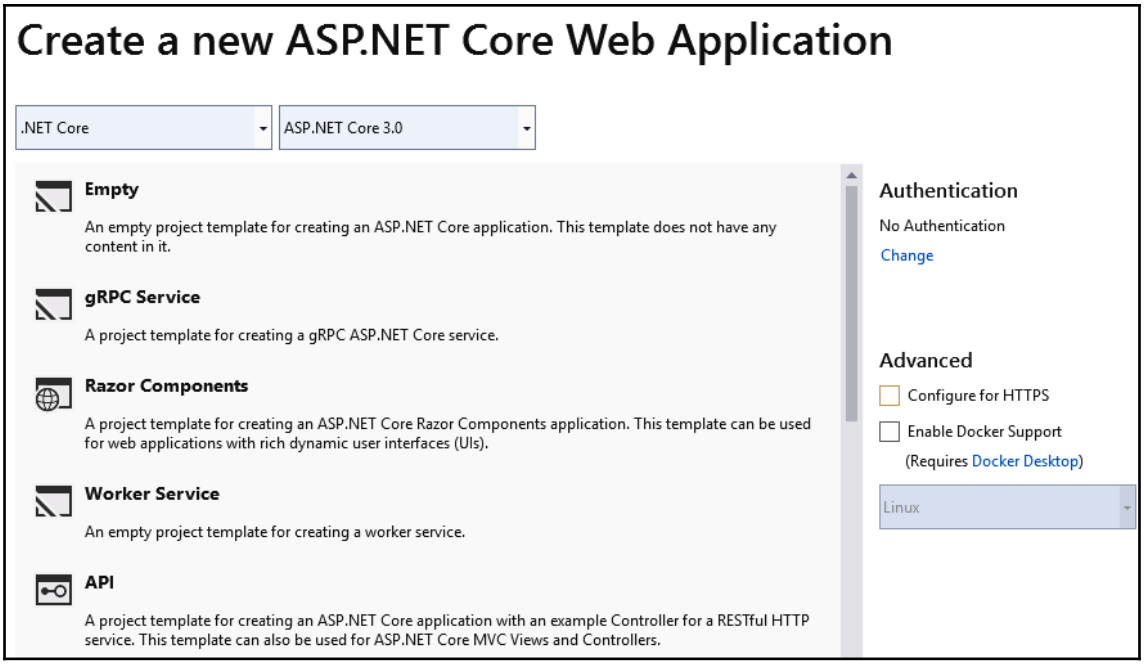
1. Start Visual Studio 2019 and click on **File** | **New** | **Project**.
2. In the **.NET Core** section, choose **ASP.NET Core Web Application**, enter the application name, the location of your repository, the solution name, and then click on **Create**:



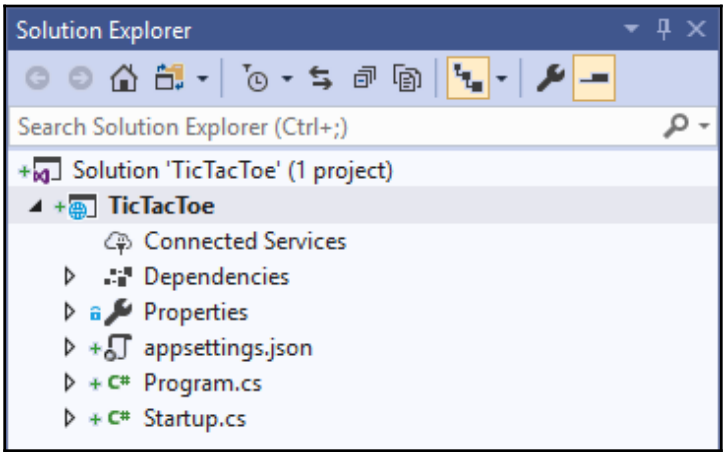
Note that if you have not created a Git repository for your application code yet, you can do it here by ticking the **Create new Git repository** checkbox.

WOW! eBook
www.wowebook.org

3. Choose the **Empty** template:



4. A new empty ASP.NET Core 3 web application project will be generated, containing only the `Program.cs` and `Startup.cs` files:



WOW! eBook
www.wowebook.org

Great! We have created our project and are now ready to implement our first feature! But, before doing that, let's take some time and see what Visual Studio 2019 has done for us behind the scenes.

Targeting different .NET Core versions in the .csproj files of your projects

For every project that Visual Studio 2019 generates, it creates a corresponding `.csproj` file, which includes several project-wide settings such as the referenced assemblies, the .NET Framework target versions, the included files, and folders, as well as multiple others.

For example, when opening the ASP.NET Core 3 project you created previously, you can see the following structure:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

You can see the `TargetFramework` setting, which allows you to define what .NET Framework versions should be included and used for building and executing the source code.

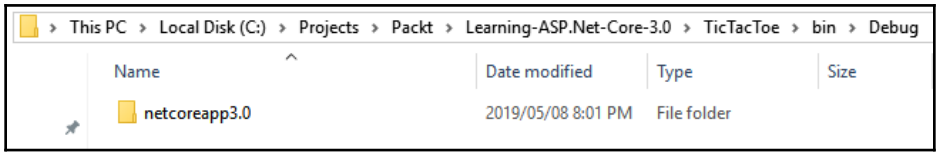
In our example, it has been set to `netcoreapp3.0`, the specific value for using the .NET Core 3 Framework:

```
<TargetFramework>netcoreapp3.0</TargetFramework>
```

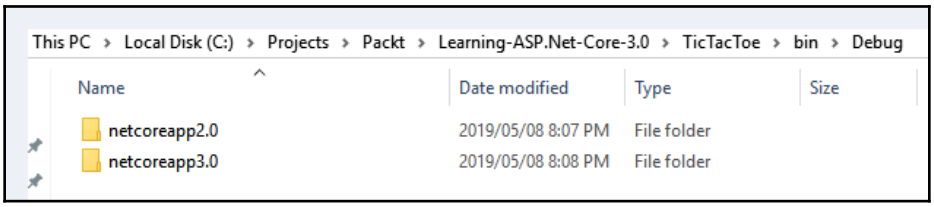
Note that you can refer to multiple .NET Framework versions within your library projects. In this case, you have to replace the `TargetFramework` element with the `TargetFrameworks` element.
For instance, if you want to cross-target .NET Core 3 and .NET Core 2, you have to use the following settings:`<TargetFrameworks>netcoreapp3.0;netcoreapp2.0</TargetFrameworks>`

WOW! eBook
www.wowebook.org

When executing your application in **Debug** mode by hitting the *F5* key, you can see that multiple folders and files have been created in the application's `Debug` folder (`/bin/Debug`):



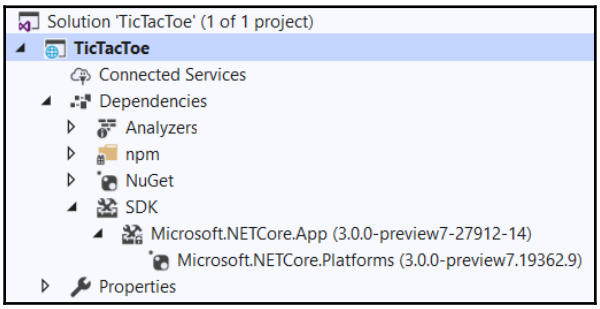
If you change the `.csproj` file and add other target frameworks, you will see that additional folders will be generated. The DLLs for each specific .NET Framework version are then put into the corresponding folders:



The preceding example uses the `TargetFrameworks` settings for .NET Core 2 and .NET Core 3.

Using the Microsoft.AspNetCore.App metapackage

When looking in Solution Explorer in the **Dependencies | SDK** section, you can see something very interesting, specific to ASP.NET Core 3 projects: the `Microsoft.AspNetCore.App` metapackage:



WOW! eBook
www.wowebook.org

The `Microsoft.AspNetCore.App` project dependency was added automatically when you created your ASP.NET Core 3 web application. This is done by default for this type of project.

However, `Microsoft.AspNetCore.App` is not a standard NuGet package, since it does not contain any code or DLLs. Instead, it acts as a metapackage, referencing other packages that it depends on. To be more specific, it includes most of the important packages for ASP.NET Core and Entity Framework Core, together with their internal and external dependencies, and takes advantage of the .NET Core runtime store.

The main difference between the new `Microsoft.AspNetCore.App` metapackage, which targets the .NET Core 3 SDK or later, and the older `Microsoft.AspNetCore.All` metapackage, which targeted previous versions of .NET Core, is the fact that the `App` metapackage no longer includes third-party dependencies, which are not primarily supported by Microsoft.

If your project is targeting `Microsoft.NET.Sdk.Web` then automatically you have access to the shared framework, and that includes packages for Application Insights, authentication, authorization, Azure App Services, and many others. In older versions of .NET Core (version 1.0 and 1.1), you had to add a horde of NuGet packages all by yourself.

Now that Microsoft has created the concept of the ASP.NET Core metapackage, you can find everything in one place. Furthermore, package trimming excludes binaries, which are not used, so that they are not published when deploying your applications.

Apart from what is already available to you with the shared framework, there are several classes that come already installed when you are using the ASP.NET Core 3 Framework, and we will take a look at them in the next section.

Introduction to the default ASP.NET Core 3 classes

When you create an ASP.NET Core 3 application from any template, be it an MVC application or an empty template, it will always have the `Program` class and the `Startup` class by default. We now take a look at the most important features in these default classes.

WOW! eBook
www.wowebook.org

ASP.NET Core 3 start up classes

For every ASP.NET Core 3 application that you will build, regardless of the template, be it MVC or indeed an empty template, there will always be minimal plumbing of classes that are used to make sure you have a working application. For an ASP.NET Core 3 application to be able to start, there are two main classes that are quite important: the `Program` class and the `Startup` class; both of these are explained in the following section.

Working with the Program class

The `Program` class is the main entry point for ASP.NET Core 3 applications. In fact, ASP.NET Core 3 applications are very similar to standard .NET Framework console applications in this regard. Both have a `Main` method that is executed when running the application.

Even the basic signature of the `Main` method, which accepts an array of strings as arguments, is the same, as you can see in the following code. To no one's surprise, this is due to the fact that an ASP.NET Core application is, in reality, a console application hosting a web application:

```
namespace TicTacToe
{
  public class Program
  {
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
  }
}
```

Normally, you do not need to touch the `Program` class in any way. By default, everything necessary to run your application is already there and pre-configured.

WOW! eBook
www.wowebook.org

However, you might want to activate some of the more advanced functionalities.

For instance, you could enable the capturing of errors during server startup and display an error page. In this case, you just have to use the following instruction:

```
webBuilder.CaptureStartupErrors(true);
```

By default, this setting is not enabled, which means that in case of errors, the host will just exit. This might not be the desired behavior and we recommend changing this parameter accordingly.

Two other useful parameters that work together are `PreferHostingUrls` and `UseUrls`. You can indicate whether the host should listen on the standard URLs defined by `Microsoft.AspNetCore.Hosting.Server.IServeror`-specific URLs you have provided. The URLs can have different formats depending on your needs, such as:

- An IPv4 address with host and port (for example, `https://192.168.57.12:5000`)
- An IPv6 address with port (for example, `https://[0:0:0:0:0:ffff:4137:270a]:5500`)
- A hostname (for example, `https://mycomputer:90`)
- A localhost (for example, `https://localhost:443`)
- A Unix socket (for example, `http://unix:/run/dan-live.sock`)

Here is an example of how you could set those parameters:

```
webBuilder.PreferHostingUrls(true);
webBuilder.UseUrls("http://localhost:5000");
```

Here is an example of a `Program` class, which includes all of the concepts shown previously:

```
public class Program
{
  public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

  public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
                webBuilder.CaptureStartupErrors(true);
```


WOW! eBook
www.wowebook.org

```
                        webBuilder.PreferHostingUrls(true);
                        webBuilder.UseUrls("http://localhost:5000");
                    });
        }
```

The default code in the `Program` class came into being from ASP.NET Core version 2.1, including version 3. Previous versions made use of `WebHostBuilder`, instead of a generic web host that we shall cover in the following section.

Working with .NET Generic Host instead of WebHostBuilder

Prior to ASP.NET Core 2.1, the `Main` method defined a host as `WebHostBuilder` as follows:

```
public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseStartup<Startup>()
                .Build();
            host.Run();
        }
```

This meant that every application was tied to be hosted as a web application. ASP.NET Core 2.1 introduced the generic host, which allows for applications that do not necessarily have to process web-based HTTP requests.

There are other applications, such as messaging and background applications, where it doesn't make sense to be tied to `WebHostBuilder` abstraction as before and, therefore, a more generic `HostBuilder` program initialization abstraction was introduced. An example of this in its raw form is shown as follows:

```
public static Task Main(string[] args)
{
    var host = new HostBuilder()
        .Build();
        host.Run();
}
```

Previous versions of ASP.NET had a `CreateWebHostBuilder()` method in the `Main` method, which is being replaced by the `CreateHostBuilder()` method in ASP.NET Core 3, and, instead of using a `WebHost.CreateDefaultBuilder(args)` method, we now have `Host.CreateDefaultBuilder(args)`.

WOW! eBook
www.wowebook.org

Working with the Startup class

Another autogenerated element, which exists in all types of ASP.NET Core 3 projects, is the `Startup` class. As you have seen previously, the `Program` class mainly handles everything associated with the hosting environment. The `Startup` class is all about the preloading and configuration of your services and middleware. Those two classes are the foundations of all ASP.NET Core 3 applications.

Let's now look at the basic structure of the `Startup` class to get a better understanding of what is provided and how to make the best use of its functionalities:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)  {   }

  public void Configure(IApplicationBuilder app,
   IHostingEnvironment env)
  {
    if (env.IsDevelopment())
    {   app.UseDeveloperExceptionPage();    }

    app.UseEndpoints(endpoints =>
      {
          endpoints.MapGet("/", async context =>
          {
              await context.Response.WriteAsync("Hello World!");
          });
      });
  }
```

There are two methods that will require your attention since you will be working with them on a fairly regular basis:

- The `ConfigureServices` method, called by the runtime and used to add services to the container
- The `Configure` method used to configure the HTTP pipeline

We said at the beginning of the chapter that we wanted more practical work, so let's get back to our Tic-Tac-Toe game and see how to use the `Startup` class in a real example!

WOW! eBook
www.wowebook.org

We are going to use MVC for implementing the application, but, since you have used the empty **ASP.NET Core 3.0 Web Application** template, nothing has been added by Visual Studio 2019 during project generation. You have to add everything by yourself; what a wonderful opportunity for a better understanding of how everything works!

The first thing to do is to add MVC to the services configuration. You do that by using the `ConfigureServices` method and just adding the MVC Middleware:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
}
```

You might say that this was too easy; so, what's the catch? There is no catch! Everything in ASP.NET Core 3 was developed around simplicity, clarity, and developer productivity.

You can see this again when configuring your MVC Middleware and setting the routing path (we will explain routing in more detail later):

```
 app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Again, we have here very clear and short instructions that make our lives as developers easier and more productive. Now is a really good time to be a developer!

In the next step, you need to enable the use of static content within your ASP.NET Core 3 application, in order to use HTML, CSS, JavaScript, and images.

Do you know how to do that? Yes, you are right; you need to add another middleware. You do that just like before by calling the corresponding `app` method:

```
app.UseStaticFiles();
```

The following is an example of a `Startup.cs` class you could use for the Tic-Tac-Toe game after having configured the various service settings seen previously:

```
public class Startup                                        {
  public void ConfigureServices(IServiceCollection services)
  { services.AddControllersWithViews();  }

  public void Configure(IApplicationBuilder app,
   IHostingEnvironment env){
    if (env.IsDevelopment())
```


WOW! eBook
www.wowebook.org

```
            app.UseDeveloperExceptionPage();
        else
            app.UseExceptionHandler("/Home/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Please note that there are many other services that you can add to the `ConfigureServices` method, examples of which include `services.AddAuthorization()`, or `services.AddAspnetCoreIdentity()`, already provided for by the ASP.NET Core Framework, and which, indeed, could be created on your own. Whatever is configured here is accessible throughout your entire application through **dependency injection** (**DI**), and we have dedicated a section on this in `Chapter 5`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 2*.

Now that we have an idea of the classes that are there by default as start up classes for an ASP.NET Core application, now is a good time to look at what a basic project structure will look like, as discussed in the following section.

Preparing the basic project structure

You will surely want to see something running after building the Tic-Tac-Toe game application. Now that we have defined how everything should work from a functional point of view, we need to start by creating the basic project structure for the application.

For ASP.NET Core 3 web applications, it is best practice to have the following structure for your projects:

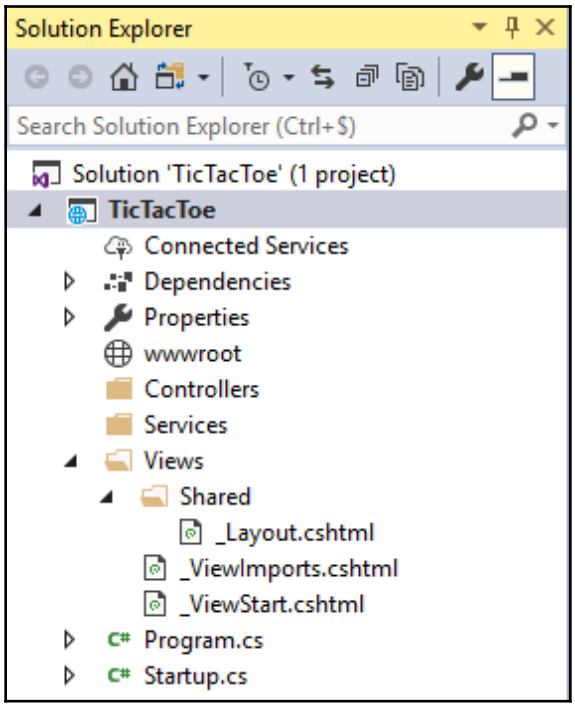
- A `Controllers` folder, containing all of the controllers of your application.
- A `Services` folder, containing all the services of your application (for example, external communication services).
- A `Views` folder, containing all the views of your application. This folder should contain a single `Shared` subfolder as well as one folder per controller.
- A `_ViewImports.cshtml` file, to define some namespaces to be available in all views.

WOW! eBook
www.wowebook.org

- A `_ViewStart.cshtml` file, to define some code to be executed at the start of each view rendering (for example, setting the layout page for all views).
- A `_Layout.cshtml` file, to define a common layout for all of your views.

Let's create the project structure:

1. Start Visual Studio 2019, open the Tic-Tac-Toe ASP.NET Core 3 project you have created, create three new folders called `Controllers`, `Services`, and `Views`, and then create a subfolder called `Shared` in the `Views` folder:



2. Create a new view page called `_ViewImports.cshtml` in the `Views` folder:

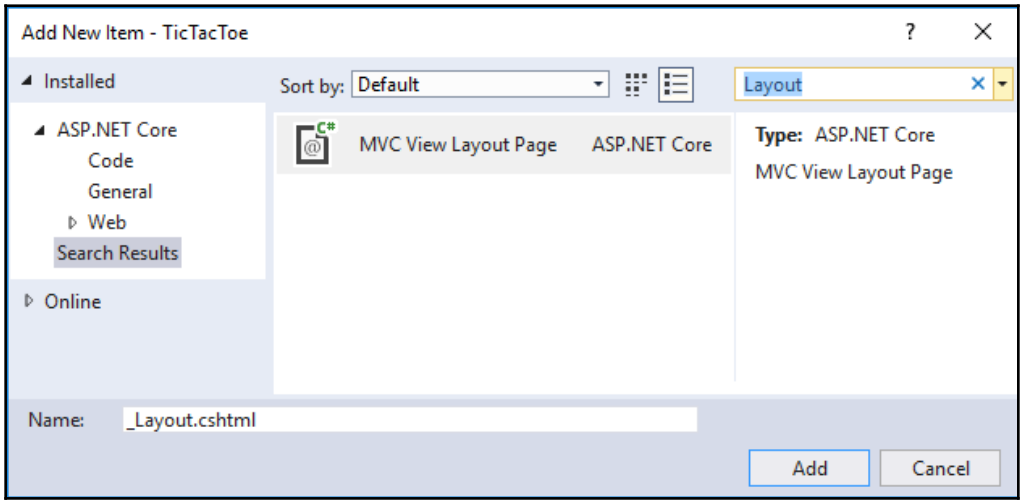
```
@using TicTacToe
@addTagHelper*, Microsoft.AspNetCore.Mvc.TagHelpers
```

3. Create a new view page called `_ViewStart.cshtml` in the `Views` folder:

```
@{ Layout = "~/Views/Shared/_Layout.cshtml"; }
```


WOW! eBook
www.wowebook.org

4. Right-click on the `Views/Shared` folder, select **Add | New Item**, enter `Layout` in the search box, select **MVC View Layout Page**, and then click on **Add**:



Note that the layout page concept will be detailed a little bit later in this chapter, but don't worry too much; it is not a very complicated concept.

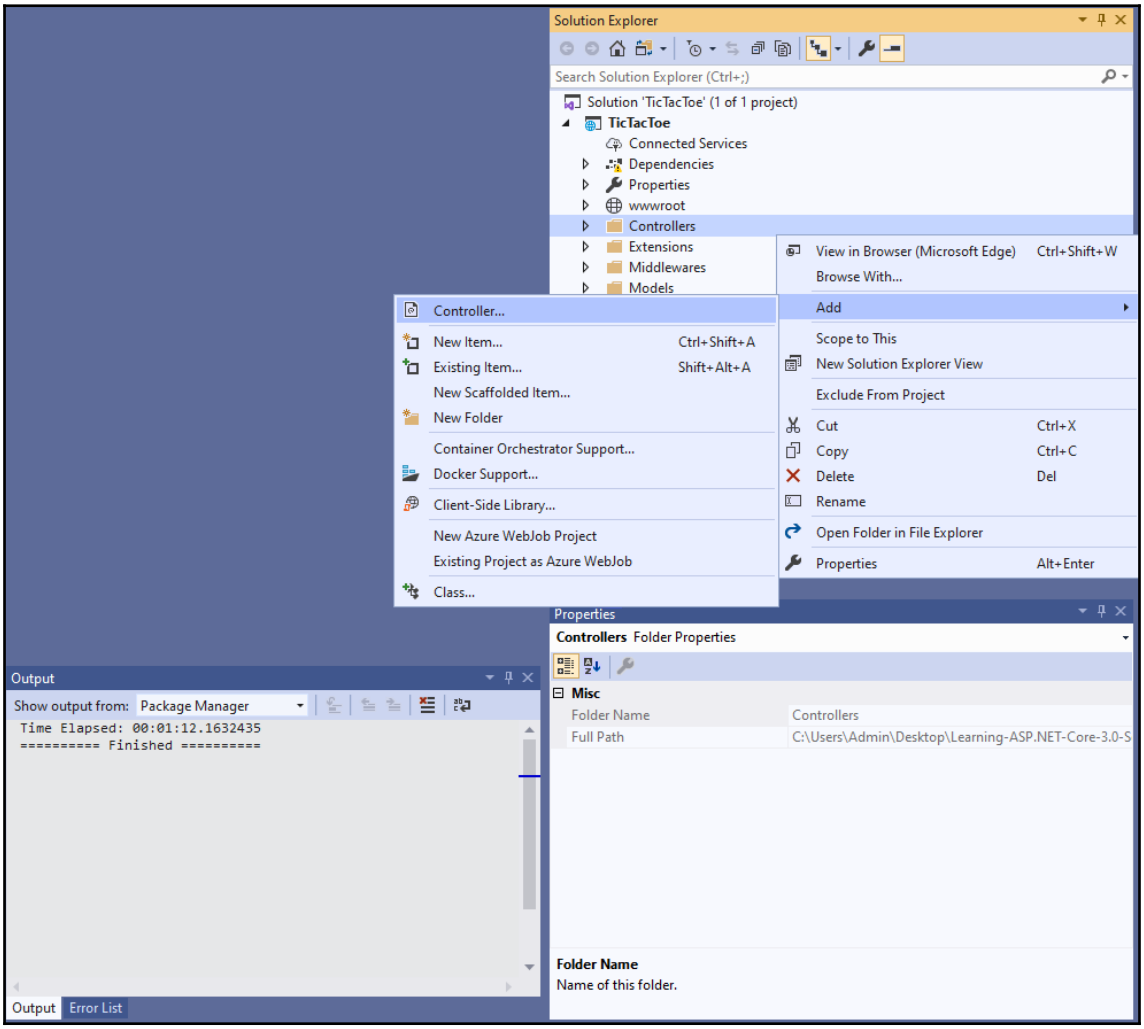
Creating the Tic-Tac-Toe home page

Since the basic project structure is now in place, we need to implement the different components that need to work together to provide the Tic-Tac-Toe game web application:

1. Update the `Program.cs` and `Startup.cs` files, as explained previously.

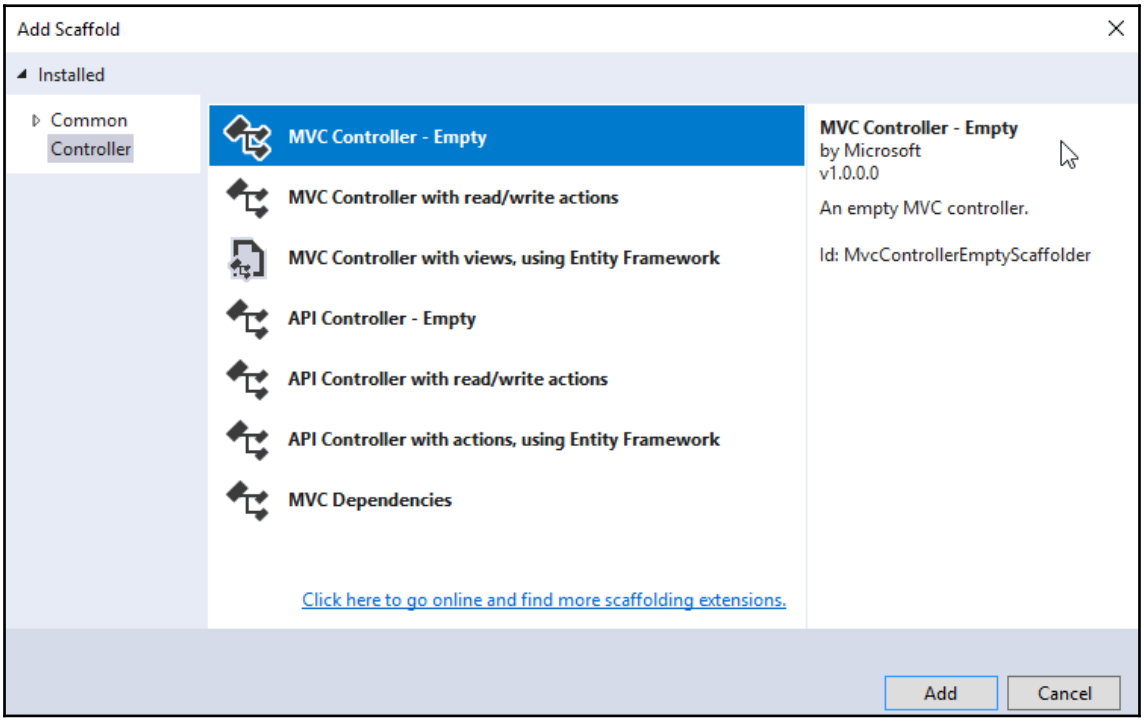
WOW! eBook
www.wowebook.org

2. Add a new controller, right-click within Solution Explorer on the `Controllers` folder, and then select **Add | Controller...**:



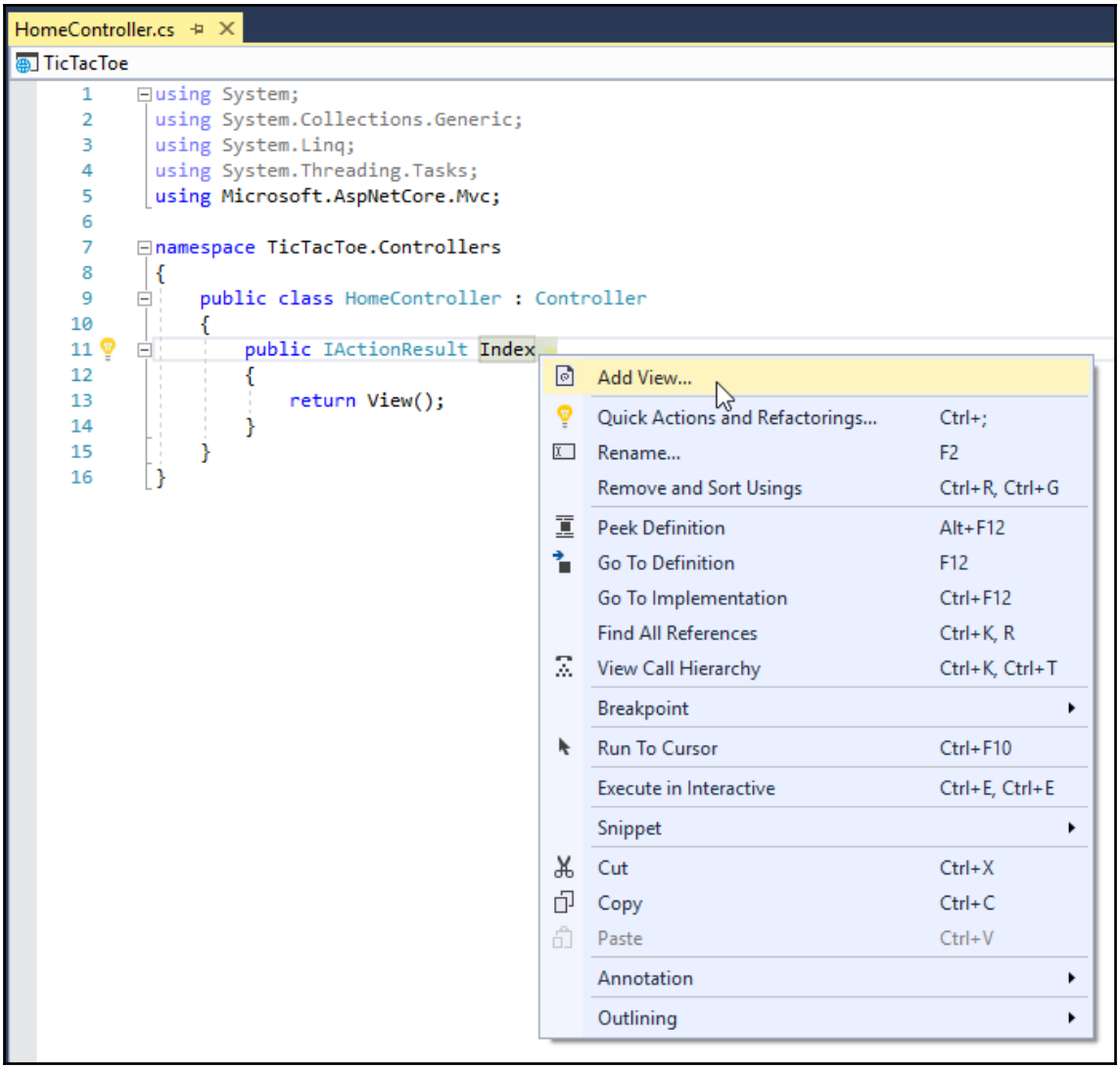
WOW! eBook
www.wowebook.org

3. In the **Add Scaffold** pop-up window, choose **MVC Controller - Empty**, and name your new controller `HomeController`:



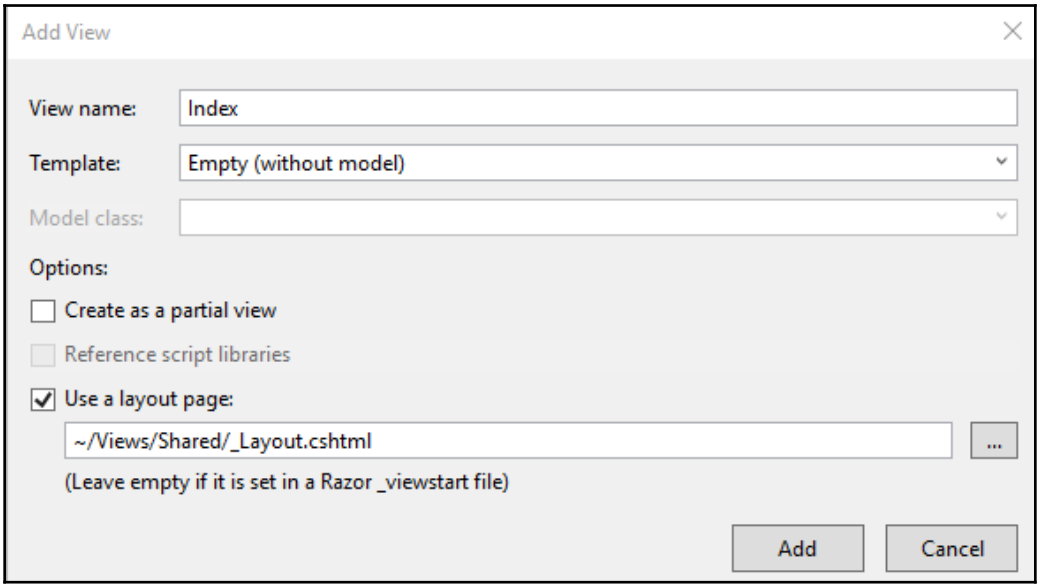
WOW! eBook
www.wowebook.org

4. Your MVC home controller gets autogenerated, containing a single method. You now need to add a corresponding view by right-clicking on the `Index` method name and selecting **Add View**... from the menu:



WOW! eBook
www.wowebook.org

5. The **Add View** window helps to define what needs to be generated. Leave the default empty template and enable the usage of the layout page we are going to modify in the next section of this chapter:



6. Congratulations! Your view gets autogenerated and you can test your application by pressing *F5*, or by clicking **Debug** on the Visual Studio 2019 menu and then **Start Debugging**. We will finalize the view later in this chapter by adding more relevant content to it.

The preceding view generated appears as plain as it could possibly be, and may not provide the best of user experiences on our web application. To help with making sure that we create more meaningful content, and systematically so, we will be using layout pages and NPM to pull in packages that will help advance the look and feel of applications that we build with ASP.NET Core 3. Let's look at how we can do that in the next section.

WOW! eBook
www.wowebook.org

Giving your web pages a more modern look by using NPM and layout pages

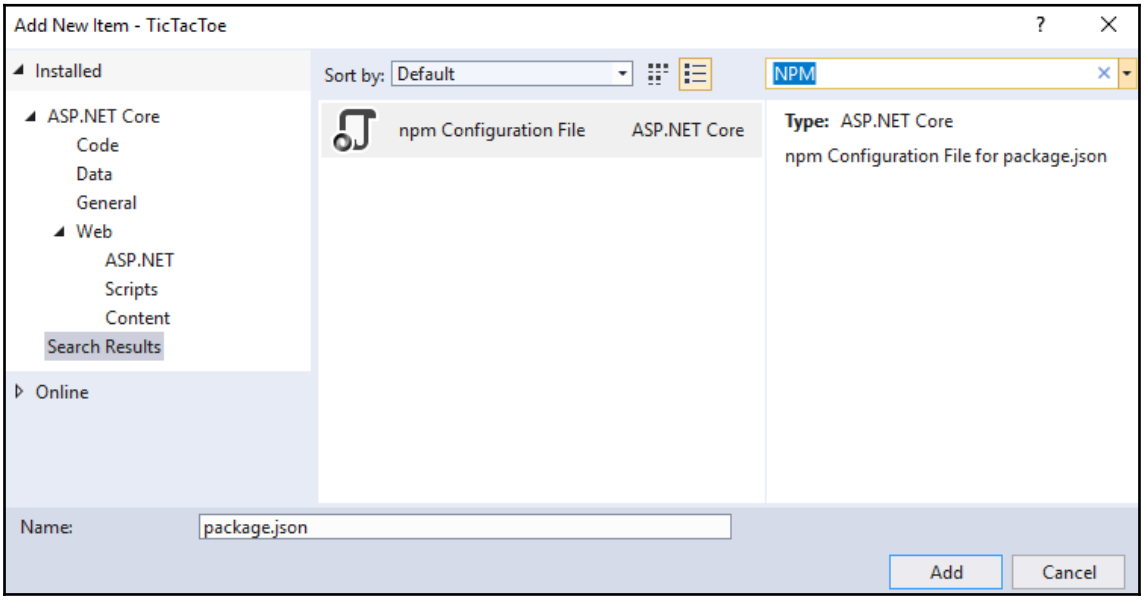
We just saw how to create a basic web page. Knowing how to do that technically is one thing, but creating web applications that succeed is a different matter entirely. It is not only about the technical implementation, but also about how to make your application visually appealing and user-friendly. While this book is not about web design and user experiences, we want to give you some quick and easy means for building better web applications in this regard.

For that, we advise using NPM (`https://www.npmjs.com/`), the most commonly used package manager on the web, in conjunction with ASP.NET Core layout pages.

NPM has had some remarkable success in the web development community in the last few years. It helps to install client-side packages with static content such as HTML, CSS, JavaScript, fonts, and images, including their dependencies.

There is some great integration and support available for NPM in Visual Studio 2019; you just have to configure it correctly in order to use it efficiently. Let's see how to do that:

1. Right-click on the Tic-Tac-Toe project, select **Add | New Item**, enter `NPM` in the search box, select **npm Configuration File**, and click on **Add**:

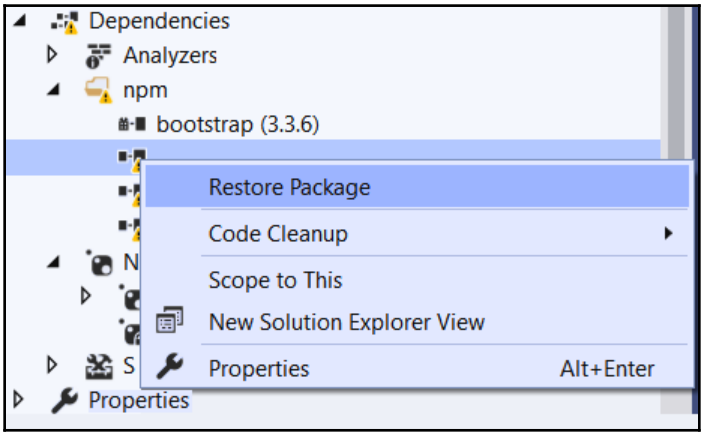


WOW! eBook
www.wowebook.org

2. Adding the **npm Configuration File** should have added a `package.json` file. Update this file with the following content:

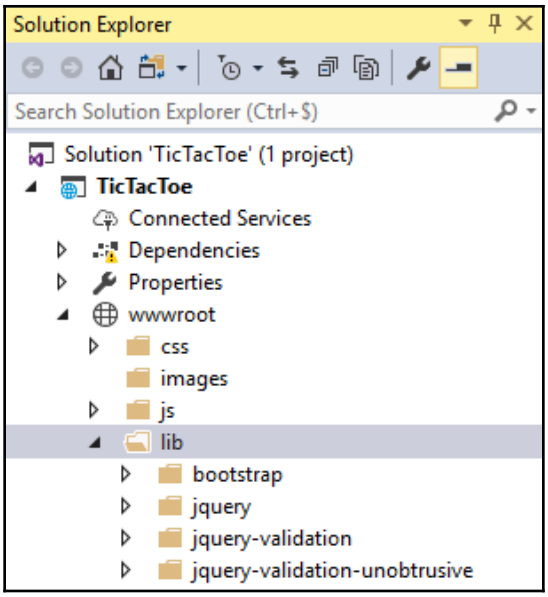
```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "bootstrap": "4.3.1",
    "jquery": "3.3.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.11",
    "popper.js": "1.14.7"
  }
}
```

3. Build the project and, following a successful build, there will be a folder named `npm`, which will be created under **Dependencies**. Right-click on **Dependencies** and then click on **Restore Package**:



4. The client-side packages (`bootstrap`, `jquery`, and more) are then downloaded into the folder you have defined and, by default, it will be (`wwwroot/lib`). The static content can now be used within your application:

WOW! eBook
www.wowebook.org



5. In the `wwwroot` folder, create a folder called `css`. Add a new style sheet called `site.css` within this folder:

```
body {
  padding-top: 50px;
  padding-bottom: 20px;
}

.body-content {
  padding-left: 15px;
  padding-right: 15px;
}

/* Set width on the form input elements since they're
 100% wide by default */
input,
select,
textarea {
  max-width: 280px;
}
```

The preceding CSS styling code allows our views to be a bit more presentable by setting padding to keep content from hitting the edges and setting a custom width that we will use for our input areas, for example, where the user will be typing their name and other details for registration.

WOW! eBook
www.wowebook.org

For validation styles, add the following code to the same `site.css` file:

```
.field-validation-error {
  color: #b94a48;
}

.field-validation-valid {
  display: none;
}

input.input-validation-error {
  border: 1px solid #b94a48; }

input[type="checkbox"].input-validation-error {
  border: 0 none;         }

.validation-summary-errors {
  color: #b94a48;         }

.validation-summary-valid {
  display: none;
}
```

This will help us to have validation messages with the right look and feel, with a customized color that will make a user automatically recognize that an error has occurred. You might have heard the term *UX Design*, and this is a simple example of **UX** (short for **user experience**) considerations that you will have to make for most applications that you will ever build.

A successful web application should have a common layout with consistent user experience when navigating from page to page. This is key for user adoption and user satisfaction. ASP.NET Core layout pages are the right solution for that.

They can be used to define templates for views in your web applications. All of your views can either use the same template or different templates that can be used depending on your specific needs.

Updating the layout page

Go to the `head` section in `_Layout.cshtml` and add the following code snippet:

```
<meta name="viewport" content="width=device-width,
     initial-scale=1.0" />
<title>@ViewData["Title"] - TicTacToe</title>
<environment include="Development">
```


WOW! eBook
www.wowebook.org

```
        <link rel="stylesheet"
         href="~/lib/bootstrap/dist/css/bootstrap.css" />
</environment>
<environment exclude="Development">
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn
        .com/bootstrap/4.3.1/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css
                /bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-
              property="position" asp-fallback-test-value="absolute"
              crossorigin="anonymous"
              integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784
              /j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"/></environment>
            <link rel="stylesheet" href="~/css/site.css" />
```

Create a `body` section with the following tags, `<body></body>` and, within the body, create a `header` navigation bar with the following code snippet:

```
<header><nav class="navbar navbar-expand-sm navbar-toggleable-sm
    navbar-light bg-white border-bottom
    box-shadow mb-3">
  <div class="container">
    <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
     action="Index">TicTacToe</a>
    <button class="navbar-toggler" type="button" data-toggle=
    "collapse" data-target=".navbar-collapse" aria-controls=
     "navbarSupportedContent" aria-expanded="false" aria-label=
     "Toggle navigation"> <span class="navbar-toggler-icon"></span>
    </button>
    <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-
     reverse">
      <ul class="navbar-nav flex-grow-1">
         <li class="nav-item">
           <a class="nav-link text-dark" asp-area="" asp-
           controller="Home" asp-    action="Index">Home</a> </li>
         <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-
            controller="Home" asp-action="Privacy">
            Privacy</a></li></ul></div></div></nav></header>
```

Within the same `body` section, outside of the navigation section, after the closing `</header>` tag, add container body content as follows:

```
 <div class="container body-content">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
    <footer class="border-top footer text-muted">
```


WOW! eBook
www.wowebook.org

```
        <div class="container">
               &copy; 2019 - TicTacToe - <a asp-area="" asp-
               controller="Home" asp-action="Privacy">Privacy</a>
        </div>
       </footer>
    </div>
```

Then, after the `body` content, add the following references at the bottom that will be used in a development environment:

```
        <environment include="Development">
          <script src="~/lib/jquery/dist/jquery.js"></script>
          <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
          <script src="~/js/site.js" asp-append-version="true"></script>
         </environment>
         <script src="~/js/site.js" asp-append-version="true"></script>
         @RenderSection("Scripts", required: false)
```

Additionally, in the case of a production environment, which we will not be using within the scope of this book, we would have something like the following code snippet:

```
       <environment exclude="Development">
           <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery
                    /3.3.1/jquery.min.js"
                   asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                   asp-fallback-test="window.jQuery"
                   crossorigin="anonymous"
                   integrity="sha256-FgpCb/KJQlLNfOu91ta32o
                   /NMZxltwRo8QtmkMRdAu8=">
           </script>
           <script src="https://stackpath.bootstrapcdn.com
           /bootstrap/4.3.1/js/bootstrap.bundle.min.js"
                   asp-fallback-src="~/lib/bootstrap/dist/js
                   /bootstrap.bundle.min.js"
                   asp-fallback-test="window.jQuery &&
                   window.jQuery.fn && window.jQuery.fn.modal"
                   crossorigin="anonymous"
                   integrity="sha384-xrRywqdh3PHs8keKZN+8zzc5TX0GRT
                   LCcmivcbNJWm2rs5C8PRhcEn3czEjhAO9o">
           </script>
       </environment>
```

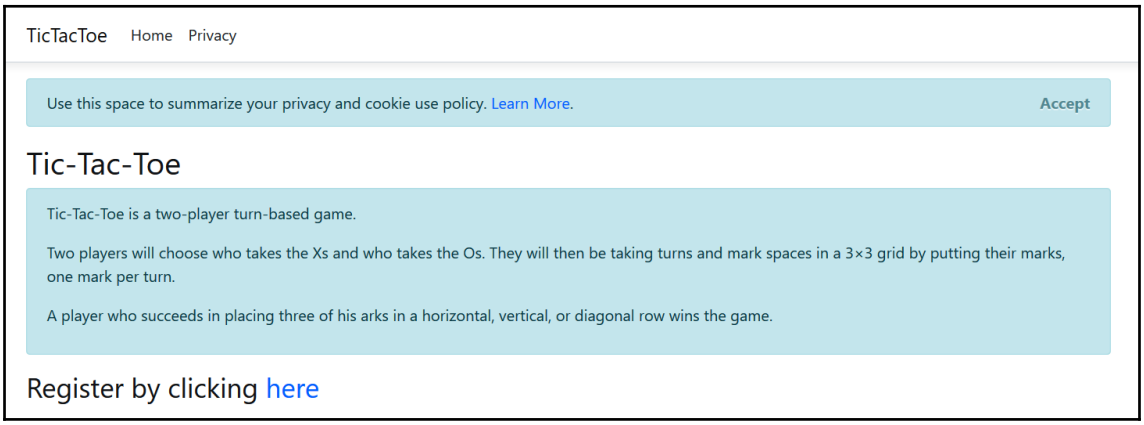
We are going to use the layout page, as updated with the preceding series of code snippets, for our sample application.

WOW! eBook
www.wowebook.org

Before creating the user registration page in the next section, let's update the home page created beforehand to show some basic information on the Tic-Tac-Toe game while using the layout page shown previously:

```
@{ ViewData["Title"] = "Home Page";
   Layout = "~/Views/Shared/_Layout.cshtml";     }
<div class="row">
 <div class="col-lg-12">
   <h2>Tic-Tac-Toe</h2>
   <div class="alert alert-info">
     <p>Tic-Tac-Toe is a two-player turn-based game.</p>
     <p>Two players will choose who takes the Xs and who takes the Os.
         They will then be taking turns and  mark spaces in a
        3×3 grid by putting their marks, one mark per turn.</p>
     <p>A player who succeeds in placing three of his arks in a
        horizontal, vertical, or diagonal row wins the
        game.</p>
    </div>
   <p><h3>Register by clicking <a asp-controller="User
    Registration"asp-view="Index">here</a></h3></p>
  </div>
</div>
```

When starting the application, you will see a new home page design, with the text that has been added earlier, as follows:



You can see in the screenshot that we have a link to allow our application users to register so that they can play our game. Therefore, we are now at a good point in terms of looking at how we can create a registration page. We will do that in the following section.

WOW! eBook
www.wowebook.org

Creating the Tic-Tac-Toe user registration page

You will now integrate the second component, the user registration page with its form, which will allow new users to register to play the Tic-Tac-Toe game:

1. Add a new folder called `Models` to the project.
2. Add a new model by right-clicking on the `Models` folder in your project and selecting **Add | Class**, and name it `UserModel`:

```
public class UserModel
{
  public Guid Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Email { get; set; }
  public string Password { get; set; }
  public bool IsEmailConfirmed { get; set; }
  public System.DateTime? EmailConfirmationDate { get;
   set; }
  public int Score { get; set; }
}
```

3. Add a new controller and call it `UserRegistrationController` (if you do not know how to do this, then refer to the *Creating the Tic-Tac-Toe home page* section).
4. Right-click on the method called `Index` and choose **Add View**. This time, select the **Create** template, choose **UserModel** as the **Model class**, as mentioned in the previous point, and enable the usage of the layout page:

WOW! eBook
www.wowebook.org

UserRegistrationController.cs

TicTacToe | TicTacToe.Controllers.UserRegistrationController

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Threading.Tasks;
5   using Microsoft.AspNetCore.Mvc;
6
7   namespace TicTacToe.Controllers
8   {
9       public class UserRegistrationController : Controller
10      {
11          public IActionResult Index()
12          {
13              return View();
14          }
15      }
16  }
```

Add View

View name: Index

Template: Create

Model class: UserModel (TicTacToe.Models)

Options:

- [ ] Create as a partial view
- [ ] Reference script libraries
- [x] Use a layout page:

~/Views/Shared/_Layout.cshtml ...

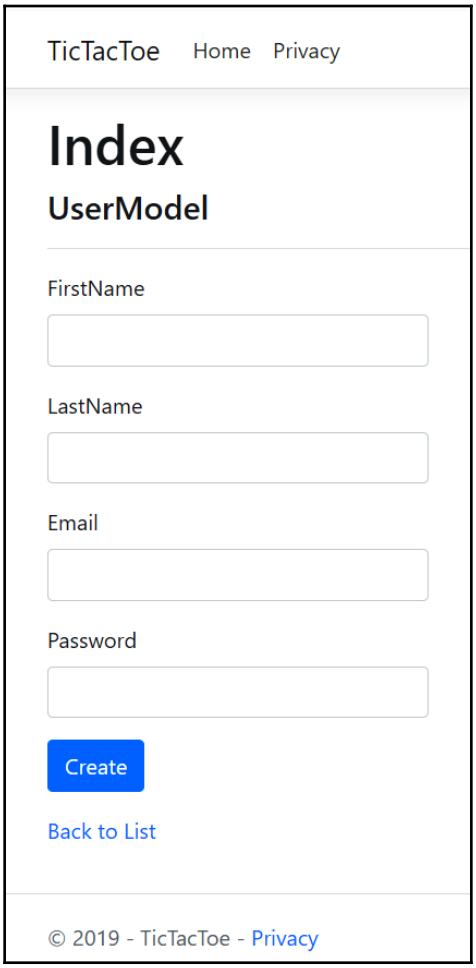
(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

Note that you can leave the layout page empty if you want to use the `_ViewStart.cshtml` file in the `Shared` folder to define a unified common layout for all your views.
The `_ViewStart.cshtml` file is used to share settings between views, while the `_ViewImports` file is used to share `using` namespaces and inject DI instances. Visual Studio 2019 includes two templates for these files.

WOW! eBook
www.wowebook.org

5. Remove the autogenerated `Id`, `IsEmailConfirmed`, `EmailConfirmationDate`, and `Score` elements from the view; we do not need them for the user registration form.
6. The view is now ready; display it by pressing *F5* and clicking on the registration link on the home page:



You will be presented with a form that can be used to fill user details such as first name, last name, email, and password. Note that the input fields are shorter; to be precise, they are 280 pixels long because of the CSS styling we did in a previous section; otherwise, they would span the entire length of the screen.

WOW! eBook
www.wowebook.org

We have done the user registration page, but you will quickly notice that a user is a central part of the application. A user will have to sign in and sign out, be validated in different ways, and will have other functionalities apart from just being registered. It is not surprising that we will need to have a user service that will co-ordinate everything that is happening for a user with the rest of the application. We are going to create a user service in the next section.

Creating the Tic-Tac-Toe user service

One of the biggest problems faced by developers while developing applications is inter-component dependencies. These dependencies make it hard to maintain and evolve your components individually because modifications may adversely impact other dependent components. For our demo application, we want to make sure that we are able to update and modify a single component or service, without needing to go and change other dependent components.

However, be assured, there are mechanisms that allow those dependencies to be broken up, one of them being DI.

While providing loose coupling, DI allows components to work together. A component only needs to know the contract implemented by another component to work with it. With a DI container, components are not directly instantiated, nor are static references used to find an instance of another component. Instead, it is the responsibility of the DI container to retrieve the correct instance during runtime.

When a component is designed with DI in mind, it is very evolutive by default and is not dependent on any other components or behaviors. For example, an authentication service can use providers for authentication that uses DI, and, if new providers are added, existing ones will not be impacted.

Using DI to encourage loose coupling

ASP.NET Core 3 includes a very simple built-in DI container, which supports constructor injection. To make a service available for the container, you have to add it within the `ConfigureService` method of the `Startup` class. Without knowing it, you have already done that before for MVC:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
}
```


WOW! eBook
www.wowebook.org

In fact, you have to do the same thing for your own custom services; you have to declare them within this method. This is really easy to do when you know what you are doing!

However, there are multiple ways of injecting your services and you need to choose which one best suits your needs:

- **Transient injection**: Creates an instance every time the method is called (for example, stateless services):

```
services.AddTransient<IExampleService, ExampleService>();
```

- **Scoped injection**: Creates an instance once per request pipeline (for example, stateful services):

```
services.AddScoped<IExampleService, ExampleService>();
```

- **Singleton injection**: Creates one single instance for the whole application:

```
services.AddSingleton<IExampleService, ExampleService>();
```

Note that you should add the instances for your services by yourself if you do not want the container to automatically dispose of them. The container will call the `Dispose` method of each service instance it creates by itself.
Here is an example of how to instantiate your services by yourself:
`services.AddSingleton(new ExampleService());`

Now that you understand how to use DI, let's apply your knowledge and create the next component for our sample application.

Creating the user service

We have created a home page as well as a user registration page. Users can click on the register link and fill out a registration form, but the form data is not yet processed in any way. We are going to add a user service that will have the responsibility of processing user-related tasks, such as user registration requests. Furthermore, you are going to apply some of the ASP.NET Core 3 DI mechanisms that you just learned:

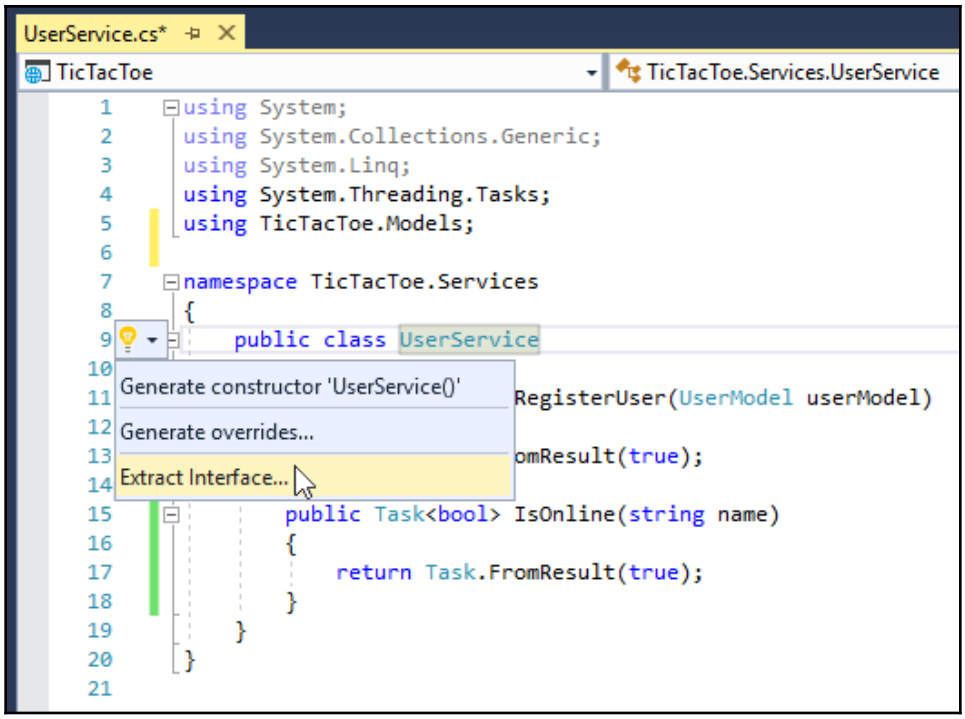
1. Add a new class called `UserService.cs` to the `Services` folder.

WOW! eBook
www.wowebook.org

2. Add new methods for user registration, and check whether a user is online, with the model created in the previous section as a parameter:

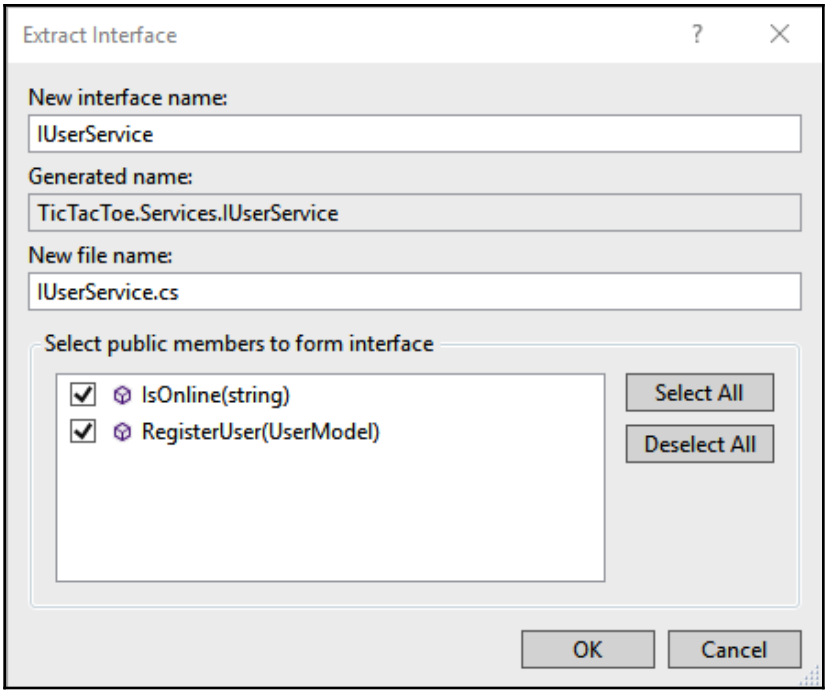
```
using TicTacToe.Models;

public class UserService
{
  public Task<bool>RegisterUser(UserModel userModel)
  {
    return Task.FromResult(true);
  }
   public Task<bool> IsOnline(string name)
    {
    return Task.FromResult(true);
    }
}
```

3. Right-click on the class and choose **Quick Actions and Refactorings**, and then click on **Extract Interface...**:



WOW! eBook
www.wowebook.org

4. Leave all of the default values in the pop-up window and click on **OK**:



5. Visual Studio 2019 will generate a new file called `IUserService.cs`, containing the extracted interface definition, as shown here:

```
public interface IUserService
{
  Task<bool>RegisterUser(UserModeluserModel);
  Task<bool> IsOnline(string name);
}
```

6. Update the `UserRegistrationController` created previously and apply the constructor injection mechanism:

```
using TicTacToe.Services;

public class UserRegistrationController : Controller
{
  private IUserService _userService;
  public UserRegistrationController(IUserService
    userService)
  {
```


WOW! eBook
www.wowebook.org

```
    _userService = userService;
  }

  public IActionResult Index()
  {
    return View();
  }
}
```

7. Add some simple code for processing the user registration within `UserRegistrationController` (we are adding validation later in the chapter):

```
[HttpPost]
public async Task<IActionResult> Index(UserModel
 userModel)
{
  await _userService.RegisterUser(userModel);
  return Content
      ($"User {userModel.FirstName} {userModel.LastName}
       has been registered successfully");
}
```

8. Go to the `Startup` class and declare `UserService` within the `ConfigureServices` method to make it available to the application:

```
using TicTacToe.Services;
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddSingleton<IUserService, UserService>();
}
```

9. Test your application by pressing *F5*, filling out the registration page, and then clicking on **OK**. You should get a `User has been registered successfully` output.

At this point, you have already created multiple components of the Tic-Tac-Toe application, and this is very good progress! Please stay focused, since the next section is very important, as it explains middleware in detail.

WOW! eBook
www.wowebook.org

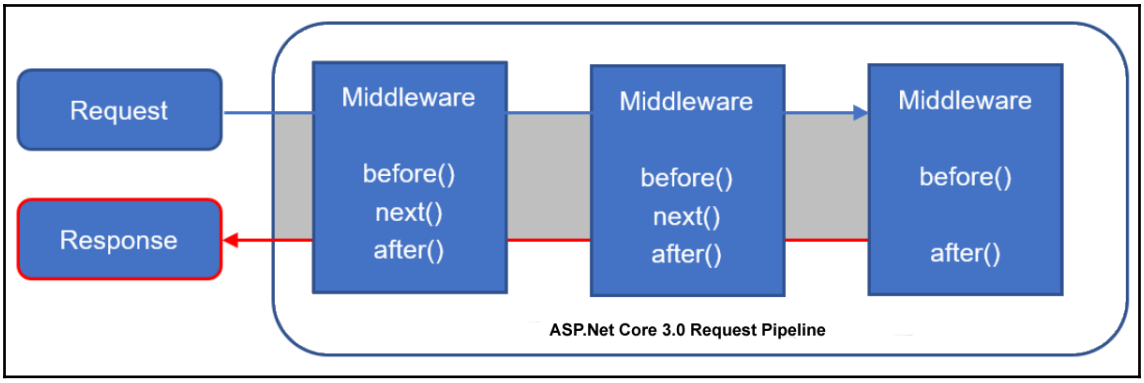
Creating a basic communication middleware for the Tic-Tac-Toe application

As you have seen before, the `Startup` class is responsible for adding and configuring middleware in your ASP.NET Core 3 applications. But what is a middleware? When and how do you use it, and how do you create your own middleware? Those are all the questions we are going to discuss now.

Essentially, multiple middleware compose the functionalities of your ASP.NET Core applications. Even the most basic functionalities, such as serving up static content, are performed by them, as you may have already noticed.

Working with middleware

Middleware are part of the ASP.NET Core 3 request pipeline for handling requests and responses. When they are chained together, they can pass incoming requests from one to another and perform actions before and after the next middleware is called within the pipeline:



Using middleware allows your applications to be more flexible and evolutive since you can add and remove middleware easily in the `Configure` method of the `Startup` class.

WOW! eBook
www.wowebook.org

Furthermore, the order in which you call the middleware in the `Configure` method is the order in which they are going to get invoked. It is advisable to call middleware in the following order so as to ensure better performance, functionality, and security:

1. Exception handling middleware
2. Static files middleware
3. Authentication middleware
4. MVC Middleware

If you do not call them in this order, you might get some unexpected behavior and even errors, since middleware actions might be applied too late or too early within the request pipeline.

For example, if you do not call the **exception handling middleware** first, you might not catch all of the exceptions that occur before its invocation. Another example is when you call the **response compression middleware** after the **static files middleware**. In this case, your static files will not be compressed, which might not be the desired behavior. So, take care of the ordering of your middleware calls; it can make a huge difference.

The following are some of the built-in middleware you can use in your applications (the list is not exhaustive; there are many more):

Authentication	OAuth 2 and OpenID authentication, based on the newest version of IdentityModel
CORS	Cross-origin resource sharing protection, based on HTTP headers
Response caching	HTTP response caching
Response compression	HTTP response gzip compression
Routing	HTTP request routing framework
Session	Basic local and distributed session object management
Static files	HTML, CSS, JavaScript, and image support including directory browsing
URL rewriting	URL SEO optimization and rewriting

The built-in middleware will be sufficient for the most basic requirements and standard use cases, but you will surely need to create your own middleware. There are two ways of doing that: creating them inline in the `Startup` class or creating them within a self-contained class.

WOW! eBook
www.wowebook.org

Let's look at how to define inline middleware first. Here are the methods available:

- `Run`
- `Map`
- `MapWhen`
- `Use`

The `Run` method is used to add middleware and immediately return a response, thus short-circuiting the request pipeline. It does not call any of the following middleware and ends the request pipeline. It is therefore advisable to place it at the end of your middleware calls (refer to middleware ordering, discussed previously).

The `Map` method allows a certain branch to be executed and the corresponding middleware to be added if the request path starts with a specific path, which means you can effectively branch the request pipeline.

The `MapWhen` method provides basically the same concept of branching the request pipeline and adding a specific middleware, but with control over the branching conditions, since it is based on the result of a `Func<HttpContext, bool>` predicate.

The `Use` method adds middleware and either allows the next middleware to be called in line or the request pipeline to be short-circuited. However, if you want to pass on the request after executing a specific action, you have to call the next middleware manually by using `next.Invoke` with the current context as a parameter.

Here are some examples of how to use these extension methods, first with `ApiPipeline` and `WebPipeline`:

```
 private static void ApiPipeline(IApplicationBuilder app)  {
  app.Run(async context =>
  {
    await context.Response.WriteAsync("Branched to Api
    Pipeline.");
  });   }

private static void WebPipeline(IApplicationBuilder app)   {
  app.MapWhen(context =>
  {
    return context.Request.Query.ContainsKey("usr");
  }, UserPipeline);

  app.Run(async context =>
  {
    await context.Response.WriteAsync("Branched to Web
```


WOW! eBook
www.wowebook.org

```
                Pipeline.");
              });     }
```

Then, there is `UserPipeline` and the `Configure` method, which makes use of the pipelines created:

```
 private static void UserPipeline(IApplicationBuilder app)     {
  app.Run(async context =>
  {
    await context.Response.WriteAsync("Branched to User
     Pipeline.");
  });    }

public void Configure(IApplicationBuilder app,
 IHostingEnvironmentenv)     {
  app.Map("/api", ApiPipeline);  app.Map("/web", WebPipeline);

  app.Use(next =>async context =>
  {
    await context.Response.WriteAsync("Called Use.");
    await next.Invoke(context);       });

  app.Run(async context =>
  {
    await context.Response.WriteAsync("Finished with Run.");
  });    }
```

As shown before, you can create your middleware inline, but this is not recommended for more advanced scenarios. We advise you to put your middleware in self-contained classes in this case, and the process for doing so is really easy. Middleware is just a class with a certain structure that is exposed via an extension method.

Creating the communication middleware

Let's perform the following steps:

1. Create a new folder called `Middleware` within your project, and then add a new class called `CommunicationMiddleware.cs`, with the following code:

```
using Microsoft.AspNetCore.Http;
using TicTacToe.Services;
public class CommunicationMiddleware
{
  private readonly RequestDelegate _next;
  private readonly IUserService _userService;
```


WOW! eBook
www.wowebook.org

```
    public CommunicationMiddleware(RequestDelegate next,
     IUserService userService)
    {
      _next = next;
      _userService = userService;
    }
    public async Task Invoke(HttpContext context)
    {
      await _next.Invoke(context);
    }
  }
```

2. Create a new folder called `Extensions` within your project, and then add a new class called `CommunicationMiddlewareExtension.cs`, with the following code:

```
using Microsoft.AspNetCore.Builder;
using TicTacToe.Middleware;
 public static class CommunicationMiddlewareExtension
 {
   public static IApplicationBuilder
    UseCommunicationMiddleware(this IApplicationBuilder app)
   {
     return app.UseMiddleware<CommunicationMiddleware>();
   }
 }
```

3. Add a `using` directive for `TicTacToe.Extensions` in the `Startup` class, and then add the communication middleware to the `Configure` method:

```
 using TicTacToe.Extensions;
 ...
 public void Configure(IApplicationBuilder app,
  IHostingEnvironment env)
 {
   ...
   app.UseCommunicationMiddleware();
   app.UseRouting();
   app.UseEndpoints(endpoints =>
   {
       endpoints.MapControllerRoute(
           name: "default",
           pattern: "{controller=Home}/{action=Index}
           /{id?}");
       endpoints.MapRazorPages();
     });
 }
```


WOW! eBook
www.wowebook.org

4. Set some breakpoints in the communication middleware implementation and start the application by pressing *F5*. You will see that the breakpoints will be hit if everything is working correctly:

```
CommunicationMiddleware.cs
TicTacToe                                   TicTacToe.Middlewares.CommunicationMiddleware
1   using Microsoft.AspNetCore.Http;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Threading;
6   using System.Threading.Tasks;
7   using TicTacToe.Services;
8
9   namespace TicTacToe.Middlewares
10  {
11      public class CommunicationMiddleware
12      {
13          private readonly RequestDelegate _next;
14          private readonly IUserService _userService;
15
16          public CommunicationMiddleware(RequestDelegate next, IUserService userService)
17          {
18              _next = next;
19              _userService = userService;
20          }
21
22          public async Task Invoke(HttpContext context)
23          {
24              await _next.Invoke(context);
25              return;
26          }
27      }
28  }
29
```

This is just a basic example of how to create your own middleware; there are no functional changes visible between this section and the others. You are going to further implement the various functionalities for finalizing the Tic-Tac-Toe application in the next chapters, and the communication middleware seen in this chapter is going to do some real work shortly.

Working with static files

When working with web applications, most of the time, you have to work with HTML, CSS, JavaScript, and images, which are considered static files by ASP.NET Core 3.

WOW! eBook
www.wowebook.org

Access to these files is not available by default, but you saw what needs to be done to allow static files to be used within your applications at the beginning of the chapter. In fact, you must add and configure the corresponding middleware to the `Startup` class to be able to serve static files:

```
app.UseStaticFiles();
```

Note that, by default, all static files served by this middleware are public and anyone can access them. If you need to protect some of your files, you need to either store them outside the `wwwroot` folder or you need to use the `FileResult` controller action, which supports the authorization middleware.

Furthermore, directory browsing is disabled by default for security reasons. You can, however, activate it easily if you need to allow users to see folders and files:

1. Add `DirectoryBrowsingMiddleware` to the `ConfigureService` method of the `Startup` class right after calling the `AddControllersWithViews()` method:

```
services.AddDirectoryBrowser();
```

2. From within the `Configure` method of the `Startup` class, call the `UseDirectoryBrowser` method (after calling the `UseCommunicationMiddleware` method) to activate directory browsing:

```
app.UseDirectoryBrowser();
```

The preceding code allows us to view the following root folders from the browser:

Index of /

Name	Size	Last Modified
css/		04/11/2017 20:25:53 +00:00
lib/		04/11/2017 14:52:24 +00:00

3. Remove the call to the `UseDirectoryBrowser` method from the `Startup` class; we do not need it for the sample application.

WOW! eBook
www.wowebook.org

Using routing, URL redirection, and URL rewriting

When building applications, routing is used to map incoming requests to route handlers (URL matching) and to generate URLs for the responses (URL generation).

The routing capabilities of ASP.NET Core 3 combine and unify the routing capabilities of MVC and web API that have existed before. They have been rebuilt from the ground up to create a common routing framework with all of the various features in a single place, available to all types of ASP.NET Core 3 projects.

Now, let's look at how routing works internally to better understand how it can be useful in your applications and how to apply it to our Tic-Tac-Toe application example.

For each request received, a matching route is retrieved, based on the request URL. Routes are processed in the order they appear within the route collection.

To be more specific, incoming requests are dispatched to the corresponding handlers. Most of the time, this is done based on data in the URL, but you could also use any data in your requests for more advanced scenarios.

If you are using the MVC Middleware, you can define and create your routes in the `Startup` class, as shown at the beginning of the chapter. This is the easiest way to get started with URL matching and URL generation:

```
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

There is also a dedicated routing middleware that you can use for working with routing in your applications, which you have seen in the previous section on middleware. You just have to add it to the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRouting();
}
```

The following is an example of how to use it to call the `UserRegistration` service in the `Startup` class.

WOW! eBook
www.wowebook.org

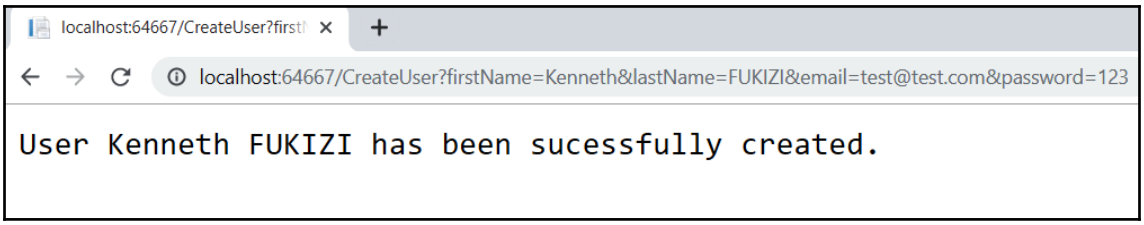
Firstly, we add `UserService` and routing to `ServiceCollection`:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddSingleton<IUserService, UserService>();
  services.AddRouting();
}
```

Then we make use of them in the `Configure` method as follows:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{  app.UseStaticFiles();
  var routeBuilder = new RouteBuilder(app);
  routeBuilder.MapGet("CreateUser", context =>
  { var firstName = context.Request.Query["firstName"];
    var lastName = context.Request.Query["lastName"];
    var email = context.Request.Query["email"];
    var password = context.Request.Query["password"];
    var userService =  context.RequestServices.
        GetService<IUserService>();
    userService.RegisterUser(new UserModel { FirstName =
    firstName, LastName = lastName, Email = email,
    Password = password });
    return context.Response.WriteAsync($"User {firstName} {lastName}
     has been successfully created.");
   });
      var newUserRoutes = routeBuilder.Build();
      app.UseRouter(newUserRoutes);
      app.UseCommunicationMiddleware();
      app.UseStatusCodePages("text/plain", "HTTP Error - Status Code:
       {0}");     }
```

If you call it with some query string parameters, you will get the following result:



Another important middleware is the **URL rewriting middleware**. It provides URL redirection and URL rewriting functionalities. However, there is a crucial difference between both that you need to understand.

WOW! eBook
www.wowebook.org

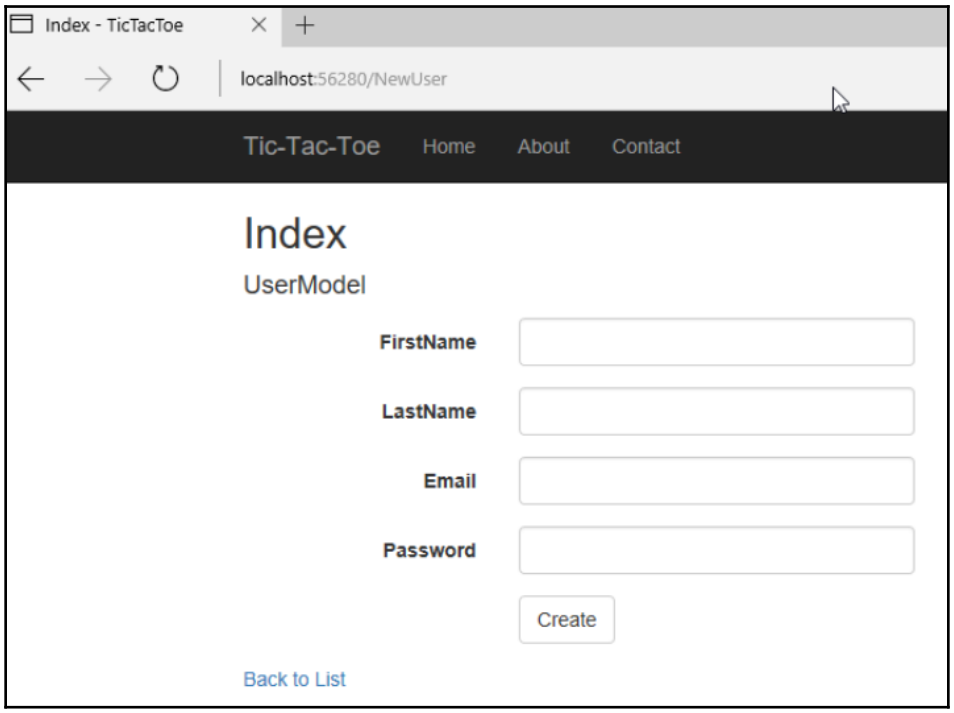
URL redirection requires a round-trip to the server and is done on the client-side. The client first receives a moved permanently `301` or moved temporary `302` HTTP status code, which indicates the new redirection URL to be used. Then, the client calls the new URL to retrieve the requested resource, so it will be visible to the client.

URL rewriting, on the other hand, is purely server-side. The server will internally retrieve the requested resource from a different resource address. The client will not know that the resource has been served from another URL as it is not visible to the client.

Coming back to the Tic-Tac-Toe application, we can use URL rewriting to give a more meaningful URL for registering new users. Instead of using `UserRegistration/Index`, we can use a much shorter URL, such as `/NewUser`:

```
var options = new RewriteOptions()
  .AddRewrite("NewUser", "/UserRegistration/Index", false);
app.UseRewriter(options);
```

Here, the user thinks that the page has been served from `/NewUser`, while, in reality, it has been served from `/UserRegistration/Index` without the user noticing:



WOW! eBook
www.wowebook.org

This comes in handy for user experience on your application, when you want the URLs to be meaningful, and can play a part in search engine optimization, where it is important for the web crawlers to match what is in the URL and the page content.

Endpoint routing for ASP.NET Core 3

Endpoint routing, which was also known as a dispatcher in its early conception, was introduced in version 2.2 of ASP.NET Core, and is, by default, recommended for ASP.NET Core 3.

If you have worked with ASP.NET Core 2.0 and earlier versions, you will find that most applications either use `RouteBuilder`, as shown in previous examples, or route attributes if you are developing APIs, which we will tackle in a later chapter. You will be familiar with `UseMVC()` and/or `UseRouter()` methods, which continue to work in ASP.NET Core 3, but endpoint routing is designed to allow developers to work with applications that are not meant to use MVC and still use routing to handle requests.

Here is an example of what you will normally find in the `Startup` class, `Configure` method, in applications prior to ASP.NET Core:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "
         {controller=Home}/{action=Index}/{id?}");
});
```

We must note that before we used `app.UseMvc`, or `app.UseRouting` in the `Configure` method, we always had to define `services.AddMvc()` in the `ConfigureServices` method.

Compare this with the default implementation in an ASP.NET Core 3 application as follows:

```
app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
```


WOW! eBook
www.wowebook.org

```
            endpoints.MapRazorPages();
            endpoints.MapControllers();
        });
```

With this implementation, we use endpoints instead of MVC, and therefore we need not add MVC specifically to the `ConfigureServices` method, and that alone makes this implementation a bit more lightweight, cutting out the overhead that comes with MVC, while it is quite important when we are building applications that do not necessarily need to follow the MVC architecture.

Our application is starting to grow, and so are the chances of encountering errors. Let's have a look at how we are going to add error handling to our application in the next section.

Adding error handling to the Tic-Tac-Toe application

When developing applications, the question is not *whether* errors and bugs will occur, but *when* they will occur. Building applications is a very complex task and it is nigh on impossible to think about all of the cases that might occur during runtime. Even if you think that you have thought about everything, then the environment is not behaving as expected; for example, a service is not available, or processing a request is taking much longer than anticipated.

You have two solutions to this problem, which need to be applied at the same time—unit tests and error handling. Unit tests will ensure correct behavior during development time from an application point of view, while error handling helps you to be prepared during runtime for environmental issues. We are going to look at how to add efficient error handling to your ASP.NET Core 3 applications in this section.

By default, if there is no error handling at all and if an exception occurs, your application will just stop, users will not be able to use it anymore, and, in the worst-case scenario, there will be an interruption of service.

WOW! eBook
www.wowebook.org

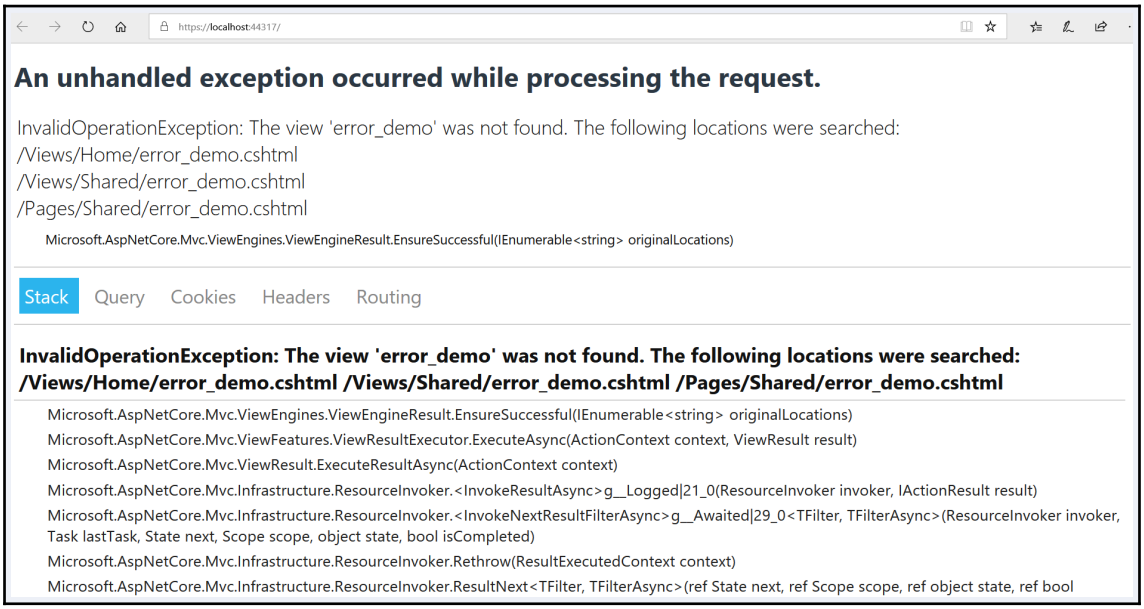
The first thing to do during development time is to activate the default development exception page; it displays detailed information on exceptions that occur. You have already seen how to do this at the beginning of the chapter:

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
}
```

On the default development exception page, you can deep dive into the raw exception details for analyzing the stack trace. You have multiple tabs that allow you to look at query string parameters, client-side cookies, and request headers.

Those are some powerful indicators that enable you to better understand what has happened and why it has happened. They should help you pinpoint problems and resolve issues more quickly during development time.

The following is an example of what happens if an exception has occurred:



WOW! eBook
www.wowebook.org

However, it is not recommended to use the default development exception page in production environments because it contains too much information about your system, which could be used to compromise your system.

For production environments, it is advised to configure a dedicated error page with static content. In the following example, you can see that the default development exception page is used during development time and that a specific error page is displayed if the application is configured to run in a non-development environment:

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
}
else
{
  app.UseExceptionHandler("/Home/Error");
}
```

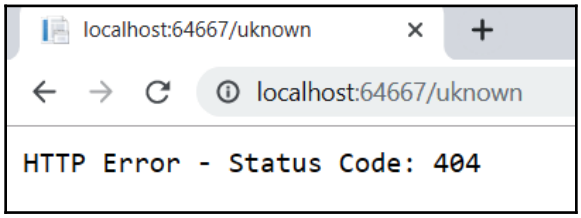
By default, no information is displayed in the case of HTTP error codes between `400` and `599`. This includes, for example, `404` (not found) and `500` (internal server error). Users will just see a blank page, which is not very user-friendly.

You should activate the specific `UseStatusCodePages` middleware in the `Startup` class. This will help you to customize what needs to be displayed in this case. Meaningful information will help users to better understand what happens within your applications and will lead to better customer satisfaction.

The most basic configuration could be to just display a text message:

```
app.UseStatusCodePages("text/plain", "HTTP Error - Status Code:
 {0}");
```

The preceding code generates the following:



WOW! eBook
www.wowebook.org

But, you can go even further. For instance, you can redirect to specific error pages for specific HTTP error status codes.

The following example shows how to send a moved temporary `302` (found) HTTP status code to the client and then redirect them to a specific error page:

```
app.UseStatusCodePagesWithRedirects("/error/{0}");
```

This example shows how to return the original HTTP status code to the client and then redirect them to a specific error page:

```
app.UseStatusCodePagesWithReExecute("/error/{0}");
```

You can disable HTTP status code pages for specific requests as shown here:

```
var statusCodePagesFeature =
   context.Features.Get<IStatusCodePagesFeature>();
   if (statusCodePagesFeature != null)
   {
     statusCodePagesFeature.Enabled = false;
   }
```

Now that we have seen how to handle errors on the outside, let's look at how to handle them on the inside, within your applications.

If we go back to the `UserRegisterController` implementation, we can see that it has multiple flaws. What if the fields have not been filled in correctly or not at all? What if the model definition has not been respected? For now, we do not require anything and we do not validate anything.

WOW! eBook
www.wowebook.org

Let's fix that and see how to build an application that is more robust:

1. Update `UserModel`, and use decorators, otherwise known as attributes, to set some properties such as `Required` and `DataType`. The `Required` attribute denotes that the following field has to have a value, and will cause an error should it not be supplied with one. The `DataType` attribute specifies that a field requires a certain data type:

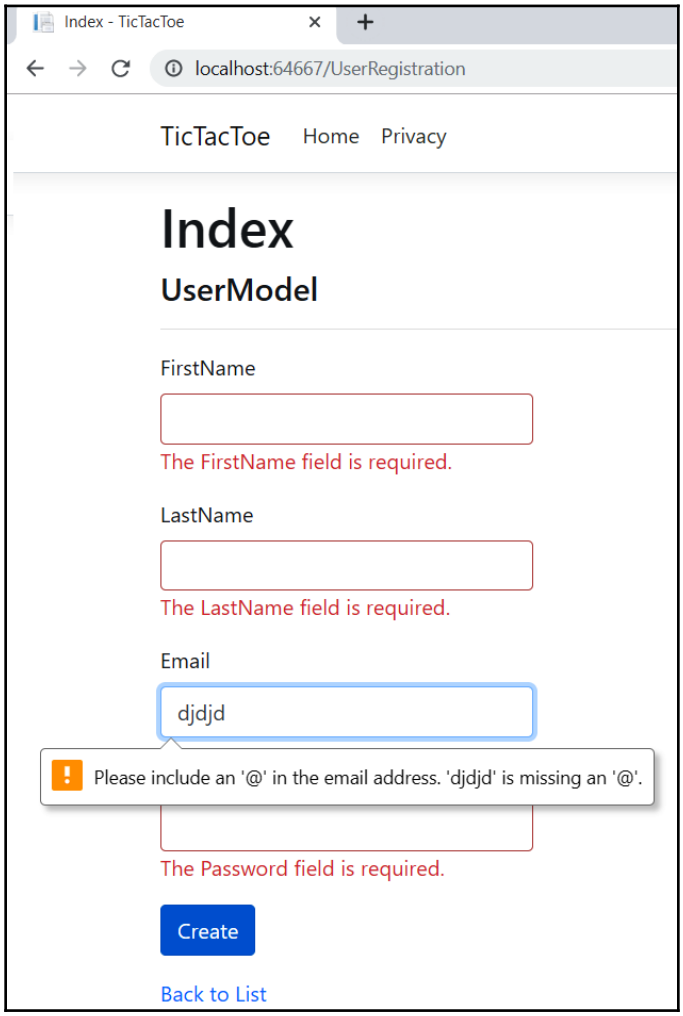
```
public class UserModel
{
  public Guid Id { get; set; }
  [Required()]
  public string FirstName { get; set; }
  [Required()]
  public string LastName { get; set; }
  [Required(), DataType(DataType.EmailAddress)]
  public string Email { get; set; }
  [Required(), DataType(DataType.Password)]
  public string Password { get; set; }
  public bool IsEmailConfirmed { get; set; }
  public System.DateTime? EmailConfirmationDate { get;
    set; }
  public int Score { get; set; }
}
```

2. Update the specific `Index` method within `UserRegistrationController`, and then add the `ModelState` validation code:

```
[HttpPost]
public async Task<IActionResult> Index(UserModel userModel)
{
  if (ModelState.IsValid)
  {
    await _userService.RegisterUser(userModel);
    return Content($"User {userModel.FirstName}
     {userModel.LastName} has been registered
       successfully");
  }
  return View(userModel);
}
```


WOW! eBook
www.wowebook.org

3. If you do not fill the required fields or you give an invalid email address and click on **OK**, you will now get a corresponding error message:



For us to reach this stage, we have gone through the process of creating a **M**odel such as the `UserModel`, a **V**iew such as the preceding one, and a **C**ontroller such as `UserRegistrationController`. In other words, we have already managed to create a functioning **MVC** application in a nutshell! Pat yourself on the back for having come this far, and expect more exciting stuff as we will be expounding more on ASP.NET Core MVC applications in the later chapters.

WOW! eBook
www.wowebook.org

Summary

In this chapter, you have learned about some of the basic concepts of ASP.NET 3. There was much to understand and much to see, and we hope you have had some fun trying everything out by yourself. You have surely made some tremendous progress!

At the beginning, you created the Tic-Tac-Toe project, and then you started implementing its different components. We explored the `Program` and `Startup` classes, saw how to use NPM and layout pages, learned how to apply DI, and used static files.

Furthermore, we introduced middleware and routing for more advanced scenarios. At the end, we illustrated how to add efficient error handling to your applications via a practical example.

In the next chapter, we will continue on and introduce additional concepts such as `WebSockets`, globalization, localization, and configuration. We will also learn how to build our application once and use the same build to run on multiple environments.

WOW! eBook
www.wowebook.org

5
Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 2

The previous chapter gave you some insights into the various functionalities and features you have at your disposal while using ASP.NET Core 3 for building efficient and more maintainable web applications. We have explained some of the basic concepts and you have seen multiple examples of how to apply them to a real-world application called **Tic-Tac-Toe**.

You have progressed quite nicely so far since you have assimilated how ASP.NET Core 3 applications are internally structured, how to configure them correctly, and how to extend them with custom behaviors, which is key for building your own applications in the future.

But let's not stop there! In this chapter, you are going to discover how to best implement the missing components, evolve the existing ones even further, and add client-side code to allow you to have a fully-running end-to-end Tic-Tac-Toe application at the end of this chapter.

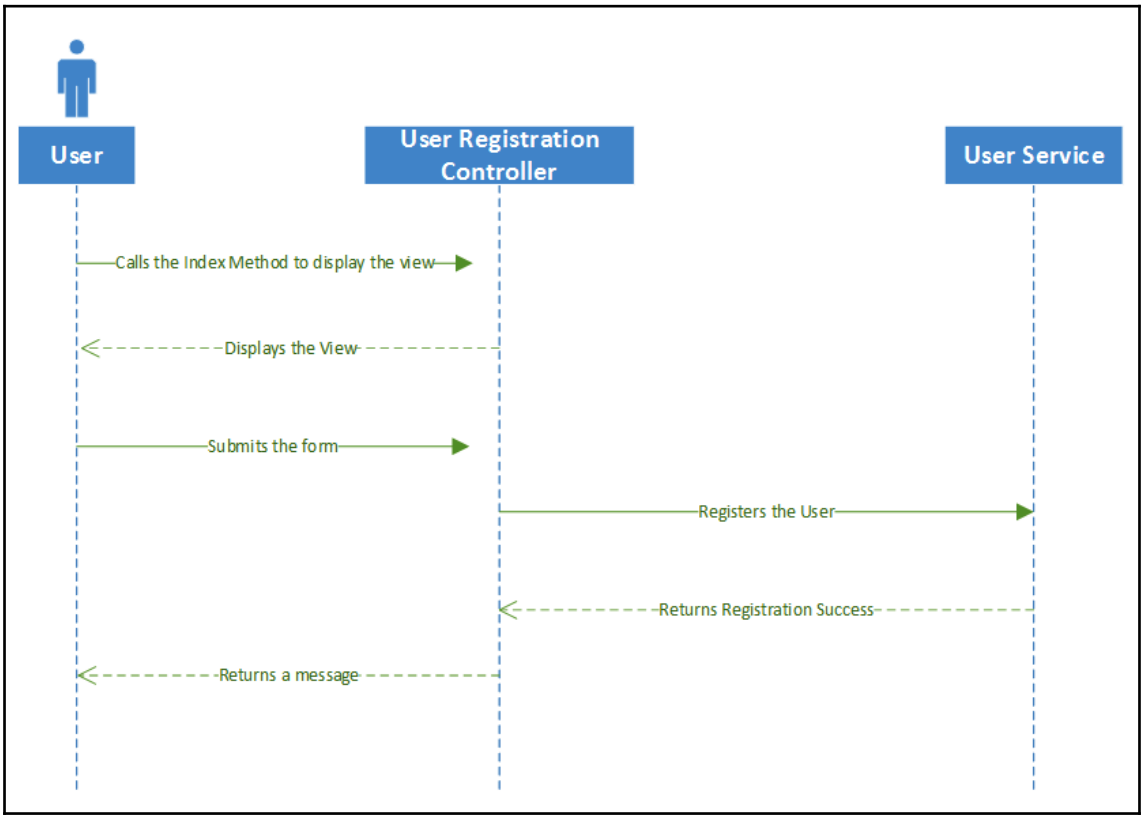
In this chapter, we will cover the following topics:

- Optimizing client-side development using JavaScript, bundling, and minification
- Working with WebSockets for real-time communication scenarios
- Taking advantage of session and user cache management
- Applying globalization and localization for multilingual user interfaces
- Configuring your applications and services
- Implementing advanced dependency injection concepts
- Building once and running on multiple environments

WOW! eBook
www.wowebook.org

Client-side development using JavaScript

In the previous chapter, you created a home page and a user registration page using the MVC pattern. You implemented a controller (`UserRegistrationController`) as well as a corresponding view for processing user registration requests. Then, you added a service (`UserService`) and middleware (`CommunicationMiddleware`), but we have only just started, so they aren't finished yet:

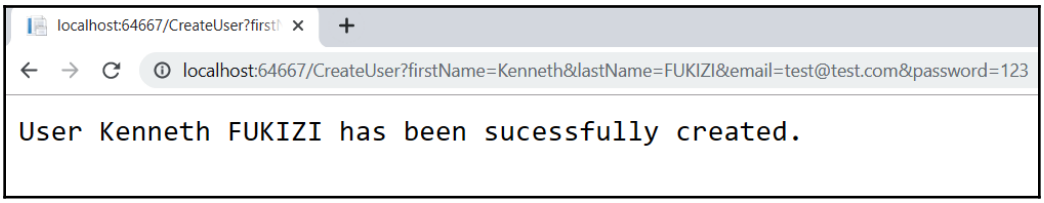


Compared to the initial workflow of the Tic-Tac-Toe application, we can see that there are still multiple things missing, such as working with the whole client-side part, actually working with the communication middleware, as well as multiple other features we still need to implement.

Let's start by working on the client-side part and learn how to apply more advanced techniques. Then, we will learn how to optimize everything as best as possible.

WOW! eBook
www.wowebook.org

As you may recall, last time, we stopped after a user had submitted their data to the registration form, which was sent to the `UserService`. Here, we just displayed a plain text message, as follows:



However, processing doesn't stop here. We need to add the whole email confirmation process using client-side development and JavaScript, and that is what we are going to do next.

Preliminary email confirmation functionality

In this section, we are going to build a tentative email confirmation functionality to demonstrate client-side development. This functionality will evolve along the course of this book and will be perfected in later chapters. But for now, let's create our email confirmation functionality as follows:

1. Start Visual Studio 2019 and open the Tic-Tac-Toe project. Add a new method called `EmailConfirmation` to `UserRegistrationController`:

```
[HttpGet]
public IActionResult EmailConfirmation (string email)
{
  ViewBag.Email = email;
  return View();
}
```

2. Right-click on the `EmailConfirmation` method, generate the corresponding view, and update it with some meaningful information:

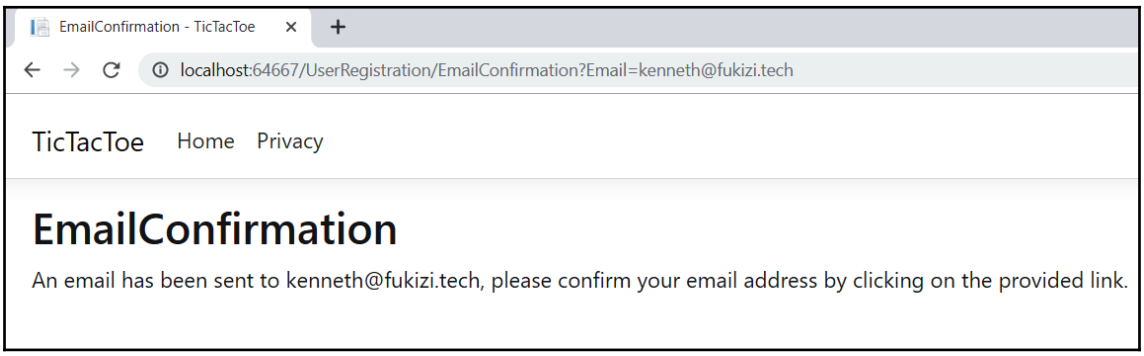
```
@{
    ViewData["Title"] = "EmailConfirmation";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>EmailConfirmation</h2>
An email has been sent to @ViewBag.Email, please
 confirm your email address by clicking on the
 provided link.
```


WOW! eBook
www.wowebook.org

3. Go to `UserRegistrationController` and modify the `Index` method to redirect to the `EmailConfirmation` method from the previous step, instead of returning the text message:

```
[HttpPost]
public async Task<IActionResult> Index(UserModel userModel)
{
  if (ModelState.IsValid)
  {
    await _userService.RegisterUser(userModel);
    return RedirectToAction(nameof(EmailConfirmation),
    new { userModel.Email });
  }
  else
  {
    return View(userModel);
  }
}
```

4. Start the application by pressing *F5*, register a new user, and verify that the new **EmailConfirmation** page is displayed correctly:



Here, you have implemented the first set of modifications in order to finalize the user registration process. In the next section, we need to check that the user has confirmed their email address.

WOW! eBook
www.wowebook.org

Email confirmation by our user

The following steps will help us check the email confirmation:

1. Add two new methods, `GetUserByEmail` and `UpdateUser`, to the `IUserService` interface. These will be used for handling the email confirmation updates:

```
public interface IUserService
{
  Task<bool> RegisterUser(UserModel userModel);
  Task<UserModel> GetUserByEmail(string email);
  Task UpdateUser(UserModel user);
}
```

2. Implement these new methods, use a static `ConcurrentBag` to persist `UserModel`, and modify the `RegisterUser` method in `UserService`, as follows:

```
public class UserService : IUserService
{
  private static  ConcurrentBag<UserModel> _userStore;
  static UserService() { _userStore = new ConcurrentBag
   <UserModel>();       }

  public Task<bool> RegisterUser(UserModel userModel) {
            _userStore.Add(userModel);
            return Task.FromResult(true);      }

  public Task<UserModel> GetUserByEmail(string email) {
            return Task.FromResult(_userStore.FirstOrDefault
             (u => u.Email == email));    }

  public  Task UpdateUser(UserModel userModel)  {
    _userStore = new ConcurrentBag<UserModel>(_userStore.Where
     (u => u.Email != userModel.Email))
            {    userModel      };
            return Task.CompletedTask;
     }
   }
```


WOW! eBook
www.wowebook.org

3. Add a new model called `GameInvitationModel` to the `Models` folder. This will be used for game invitations after successful user registration:

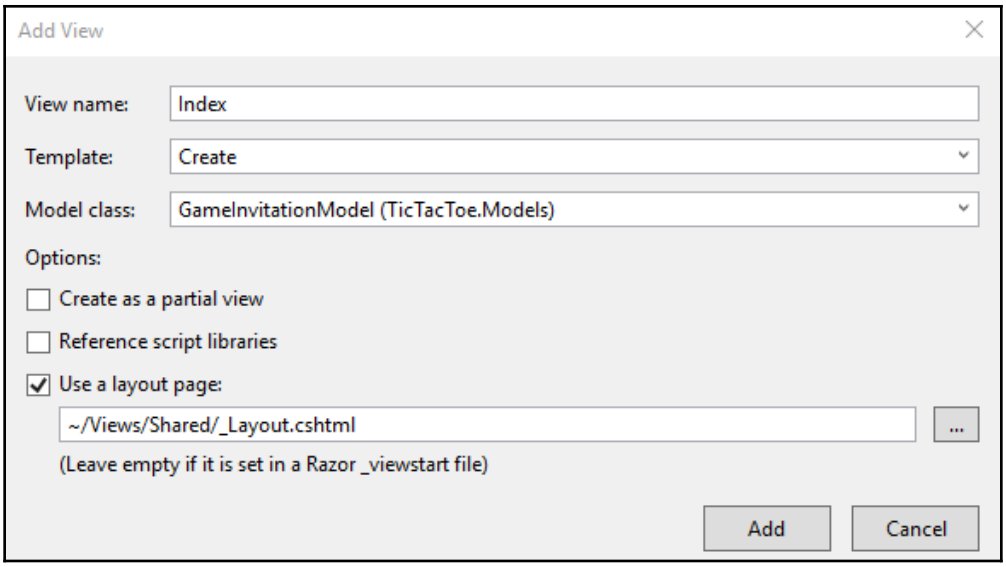
```
public class GameInvitationModel
{
  public Guid Id { get; set; }
  public string EmailTo { get; set; }
  public string InvitedBy { get; set; }
  public bool IsConfirmed { get; set; }
  public DateTime ConfirmationDate { get; set; }
}
```

4. Add a new controller called `GameInvitationController` and update its `Index` method to automatically set the `InvitedBy` property:

```
using TicTacToe.Services;
public class GameInvitationController : Controller
{
  private IUserService _userService;
  public GameInvitationController(IUserService userService)
  {
     _userService = userService;
  }

  [HttpGet]
  public async Task<IActionResult> Index(string email)
  {
    var gameInvitationModel = new GameInvitationModel
     {InvitedBy = email };
    return View(gameInvitationModel);
    }
  }
```

5. Generate a corresponding view by right-clicking on the `Index` method, selecting the **Create** template, and selecting `GameInvitationModel(TicTacToe.Models)` as the **Model class**, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



6. Modify the auto-generated view and remove all the unnecessary input controls, except the `EmailTo` input control:

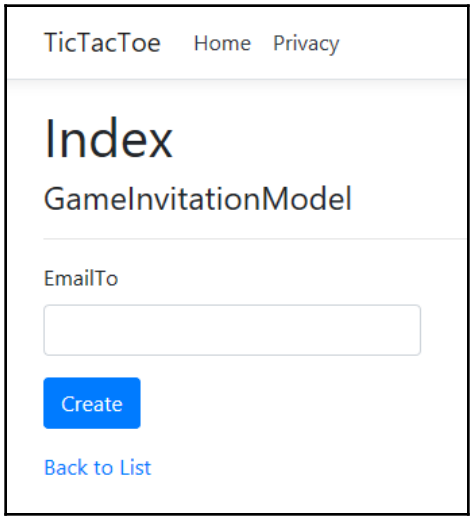
```
@model TicTacToe.Models.GameInvitationModel

<h4>GameInvitationModel</h4>
<div class="row">
  <div class="col-md-4">
    <form asp-action="Index">
       <div asp-validation-summary="ModelOnly" class="text
        -danger"></div>
         <div class="form-group">
            <label asp-for="EmailTo" class="control-label"></label>
            <input asp-for="EmailTo" class="form-control" />
            <span asp-validation-for="EmailTo" class="text
             -danger"></span>
         </div>
         <div class="form-group">
            <input type="submit" value="Create" class="btn btn-
             primary" />
         </div>
        </form>
    </div>
</div>
```


WOW! eBook
www.wowebook.org

7. Now, update the `EmailConfirmation` method in `UserRegistrationController`. The user has to be redirected to `GameInvitationController` after their email has been confirmed. As you can see, we are going to simulate this confirmation in the code for now:

```
[HttpGet]
public async Task<IActionResult> EmailConfirmation(
  string email)
{
  var user = await _userService.GetUserByEmail(email);
  if (user?.IsEmailConfirmed == true)
    return RedirectToAction("Index", "GameInvitation",
     new { email = email });

  ViewBag.Email = email;
  user.IsEmailConfirmed = true;
  user.EmailConfirmationDate = DateTime.Now;
  await _userService.UpdateUser(user);
  return View();
}
```

8. Start the application by pressing *F5*, register a new user, and verify that the **EmailConfirmation** page is displayed. In Microsoft Edge, press *F5* to reload the page, and, if everything is working as expected, you should be redirected to the game invitation page, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

Great—some more progress! Everything is working up until the game invitation now, but unfortunately, user intervention is still necessary. The user has to manually refresh the email confirmation page by pressing *F5* until their email has been confirmed; only then will they be redirected to the game invitation page.

The entire refresh process must be automated and optimized in the next step. Your options are as follows:

- Place an HTML `meta` refresh tag in the `head` section of the page.
- Use simple JavaScript, which does the refresh programmatically.
- Implement **XMLHttpRequest** (**XHR**) using jQuery.

HTML5 has introduced the meta refresh tag for automatically refreshing pages after a certain amount of time. However, this method is not advisable because it creates a high server load. Also, the security setting in Microsoft Edge may completely deactivate it and some ad blockers will stop it from working. So, if you use it, you cannot be sure that it is going to work correctly.

Using simple JavaScript may very well automate the page refresh programmatically, but it has mainly the same flaws and so it isn't recommended either.

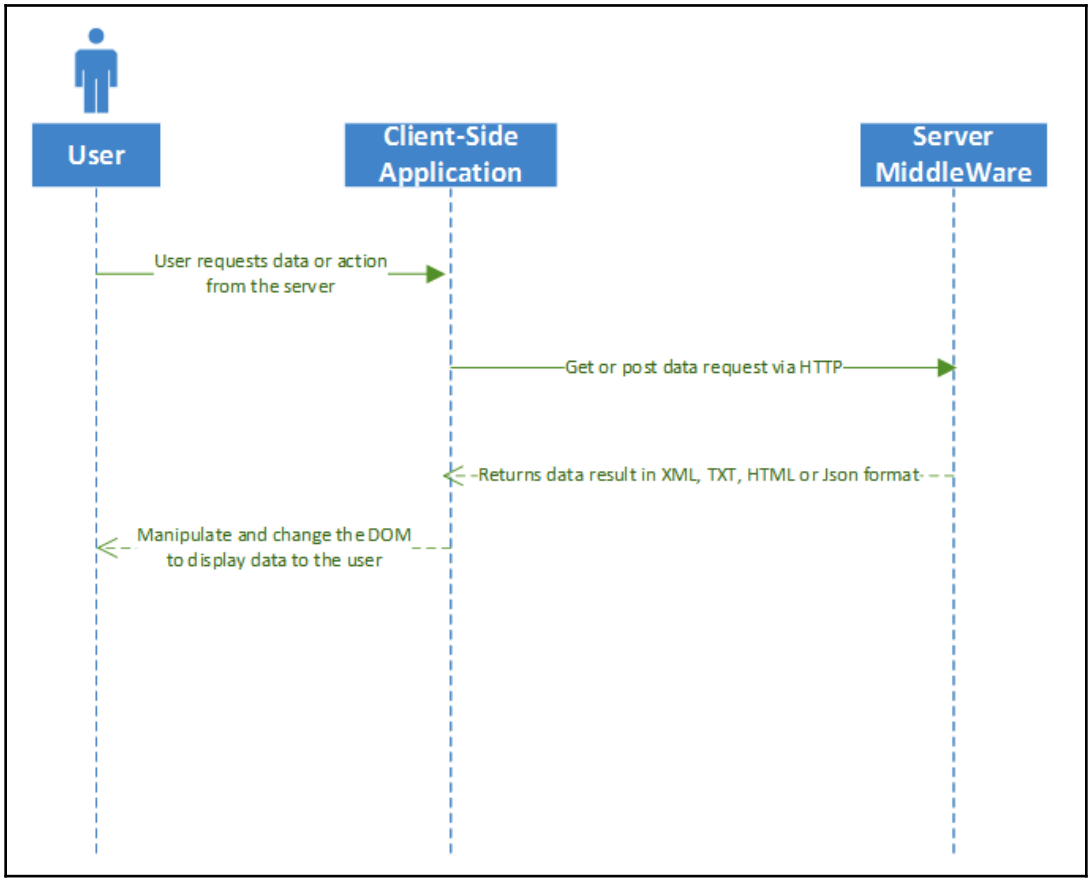
Using XMLHttpRequest

We've just mentioned that it's not recommended to use simple JavaScript and `meta` refresh tags for the refreshing process, so let's introduce XHR, which is what we are really looking for. It provides exactly what we need for our Tic-Tac-Toe application as it allows the following:

- Updating web pages without reloading them
- Requesting and receiving data from the server, even after page load
- Sending data to the server in the background

WOW! eBook
www.wowebook.org

This can be seen in the following diagram:



Now, you are going to use XHR for automating and optimizing the client-side implementation of the user registration email confirmation process. The steps for doing so are as follows:

1. Create a new folder called `app` in the `wwwroot` folder (this folder will contain all the client-side code shown in the following steps) and create a subfolder within this folder called `js`.
2. Add a new JavaScript file called `scripts1.js` to the `wwwroot/app/js` folder that contains the following content:

```
var interval;
function EmailConfirmation(email) {
  interval = setInterval(() => {
```


WOW! eBook
www.wowebook.org

```
    CheckEmailConfirmationStatus(email);
  }, 5000);
}
```

3. Add a new JavaScript file called `scripts2.js` to the `wwwroot/app/js` folder that contains the following content:

```
function CheckEmailConfirmationStatus(email) {
  $.get("/CheckEmailConfirmationStatus?email=" + email,
    function (data) {
      if (data === "OK") {
        if (interval !== null)
        clearInterval(interval);
        alert("ok");
      }
  });
}
```

4. Open the layout page in the `Views\Shared\_Layout.cshtml` file and add a new development environment element before the closing `body` tag (it is good practice to put it here):

```
<environment include="Development">
  <script src="~/app/js/scripts1.js"></script>
  <script src="~/app/js/scripts2.js"></script>
</environment>
```

5. Update the `Invoke` method in the communication middleware and add a new await to access a `ProcessEmailConfirmation` method called `ProcessEmailConfirmation`:

```
public async Task Invoke(HttpContext context)
{
  if (context.Request.Path.Equals(
    "/CheckEmailConfirmationStatus"))
  {
    await ProcessEmailConfirmation(context);
  }
  else
  {
    await _next?.Invoke(context);
  }
}
```


WOW! eBook
www.wowebook.org

The `ProcessEmailConfirmation` method is going to simulate email confirmation functionality. We define the method as follows:

```
private async Task ProcessEmailConfirmation(  HttpContext
 context)
{
  var email = context.Request.Query["email"];
  var user = await _userService.GetUserByEmail(email);

  if (string.IsNullOrEmpty(email))
  {  await context.Response.WriteAsync("BadRequest:Email is
    required"); }

  else if ((await _userService.GetUserByEmail
    (email)).IsEmailConfirmed)
  {  await context.Response.WriteAsync("OK");  }
  else
  {
    await context.Response.WriteAsync(
     "WaitingForEmailConfirmation");
    user.IsEmailConfirmed = true;
    user.EmailConfirmationDate = DateTime.Now;
    _userService.UpdateUser(user).Wait();
  }
}
```

6. Update the `EmailConfirmation` view, under the `UserRegistration` folder, by adding a call to the JavaScript `EmailConfirmation` function (from the previous step) at the bottom of the page, as follows:

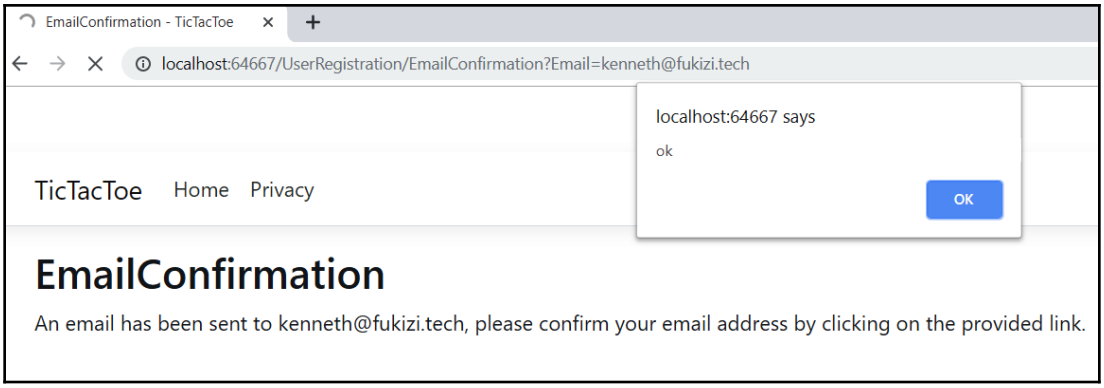
```
@section Scripts
{
  <script>
     $(document).ready(function () {
       EmailConfirmation('@ViewBag.Email');
     });
  </script>
}
```

7. Update the `EmailConfirmation` method in `UserRegistrationController`. Since the communication middleware is going to simulate the effective email confirmation, remove the following lines:

```
user.IsEmailConfirmed = true;
user.EmailConfirmationDate = DateTime.Now;
await _userService.UpdateUser(user);
```


WOW! eBook
www.wowebook.org

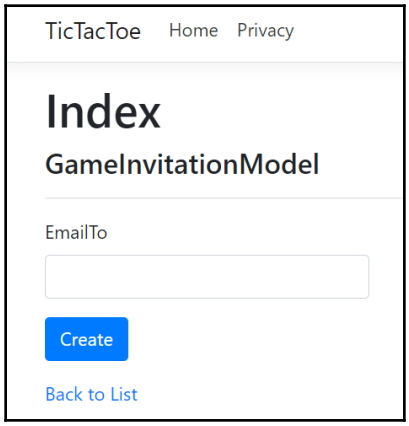
8. Start the application by pressing *F5* and register a new user. You will see a JavaScript alert box returning `WaitingForEmailConfirmation` and, after some time, another with **ok**:



9. Now, you have to update the `CheckEmailConfirmationStatus` method in the `scripts2.js` file to redirect our users to the game invitation page, in case of a confirmed email. For that, remove the `alert("OK");` instruction and add the following instruction in its place:

```
window.location.href = "/GameInvitation?email=" + email;
```

10. Start the application by pressing *F5* and register a new user. Everything should be automated and you should be automatically redirected to the game invitation page at the end:

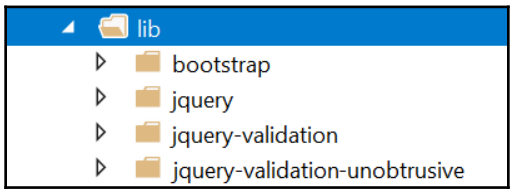


WOW! eBook
www.wowebook.org

Note that, if you still see the alert box, even though you have updated the project in Visual Studio, you might have to delete the cached data in your browser to have the JavaScript refreshed correctly in your browser and see the new behavior.

Optimizing your web applications and using bundling and minification

As you saw in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, we have chosen the community-proven **Node Package Manager** (**NPM**) as a client-side package manager. We have left the `appsettings.json` file untouched, which means that we have restored the four default packages and added some references within the ASP.NET Core 3 layout page to use them:



In today's world of modern web application development, it is good practice to separate client-side JavaScript code and CSS style sheets into multiple files during development. However, having so many files may lead to performance and bandwidth problems during runtime in production environments.

That is why, during the build process, everything must be optimized before generating the final release packages, which means that JavaScript and CSS files must be bundled and minified. TypeScript and CoffeeScript files must be transcompiled into JavaScript.

Bundling and minification are two techniques you can use to improve the overall page load performance of your web applications. Bundling allows you to combine multiple files into a single file, whereas minification optimizes the code of your JavaScript and CSS files for smaller payloads. They work together to reduce the number of server requests, as well as the overall request size.

WOW! eBook
www.wowebook.org

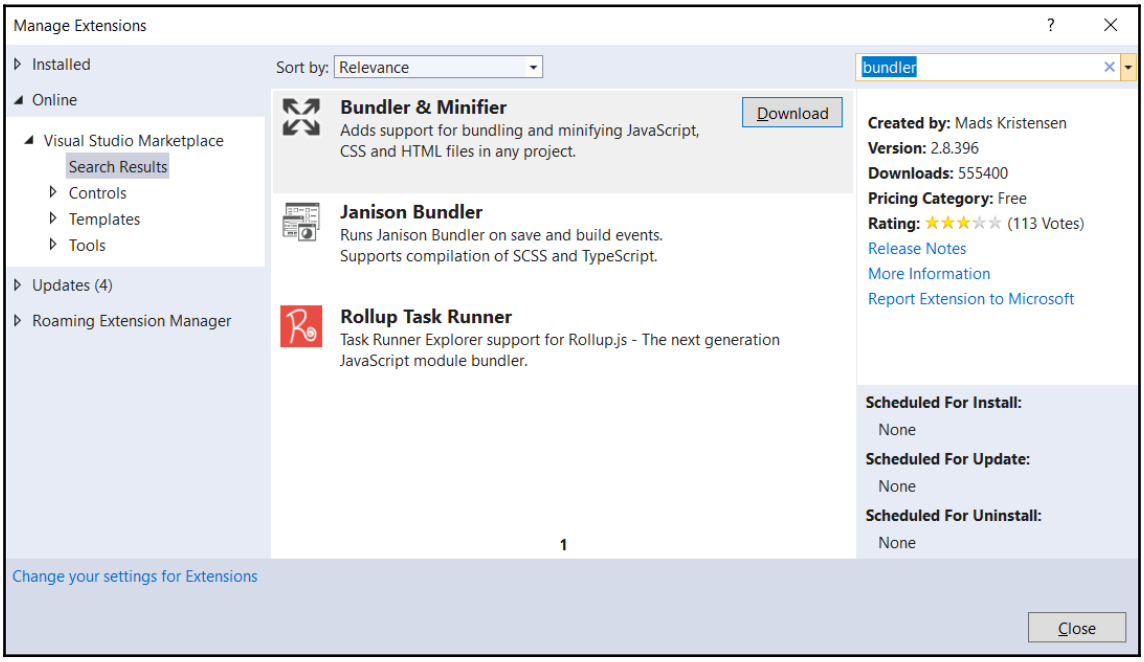
ASP.NET Core 3 supports different solutions for bundling and minification:

- Visual Studio Bundler & Minifier extension
- Gulp
- Grunt

Bundling and minification in action

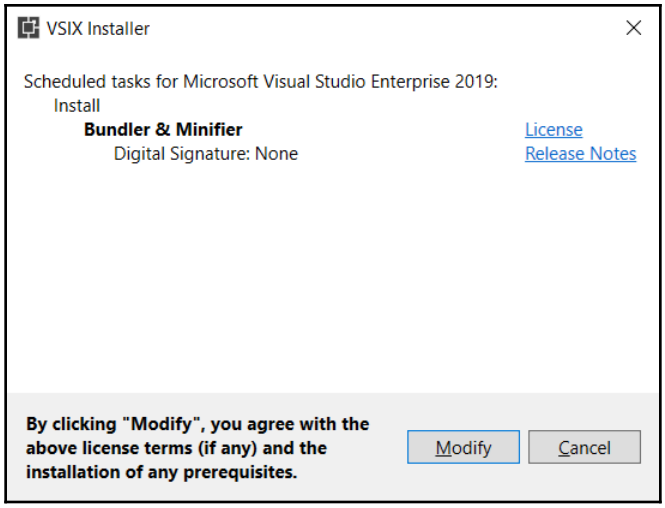
Let's learn how to bundle and minify multiple JavaScript files in the Tic-Tac-Toe project by using the Visual Studio Bundler & Minifier extension with the `bundleconfig.json` file:

1. In the top menu, select **Extensions**, click on **Online**, enter `Bundler` in the search box, select **Bundler & Minifier**, and click on **Download**:



WOW! eBook
www.wowebook.org

2. Close Visual Studio; the installation will continue. Next, click on **Modify**:

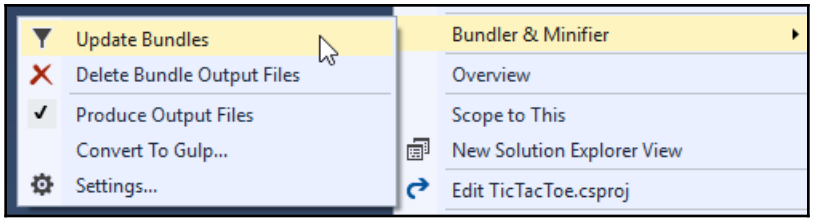


3. Restart Visual Studio. Now, you are going to optimize the number of opened connections, as well as the bandwidth usage by bundling and minifying. For that, add a new JSON file called `bundleconfig.json` to the project.
4. Update the `bundleconfig.json` file so that you can bundle the two JavaScript files into a single one called `site.js` and to minify the `site.css` and `site.js` files:

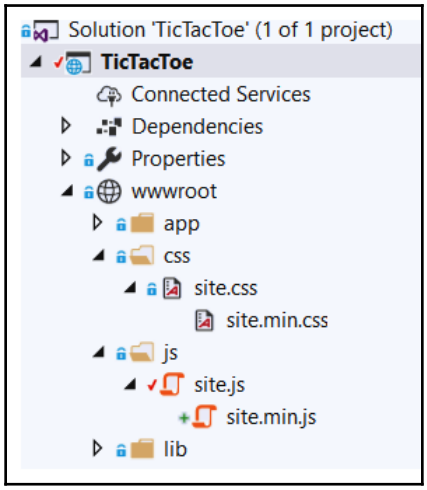
```
[
  {"outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"  ]},
  {"outputFileName": "wwwroot/js/site.js",
    "inputFiles": [
      "wwwroot/app/js/scripts1.js",
      "wwwroot/app/js/scripts2.js"  ],
    "sourceMap": true,
    "includeInProject": true   },
  {"outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": ["wwwroot/js/site.js"],
    "minify": {
      "enabled": true,
      "renameLocals": true },
    "sourceMap": false    }
]
```


WOW! eBook
www.wowebook.org

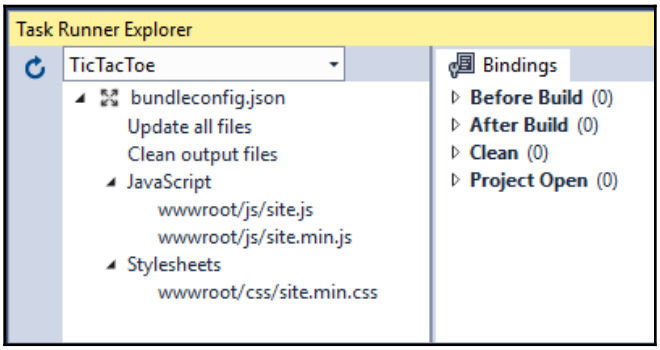
5. Right-click on the project and select **Bundler & Minifier | Update Bundles**:



6. When looking in the Solution Explorer, you will see that two new files called `site.min.css` and `site.min.js` have been generated:

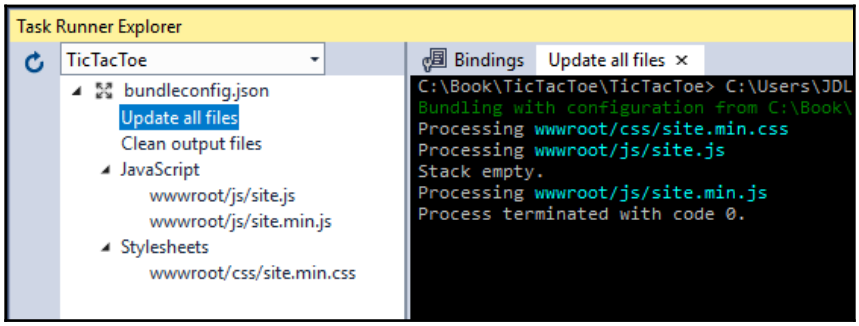


7. When looking in the **Task Runner Explorer**, you will see the bundling and minifying process you have configured for the project:

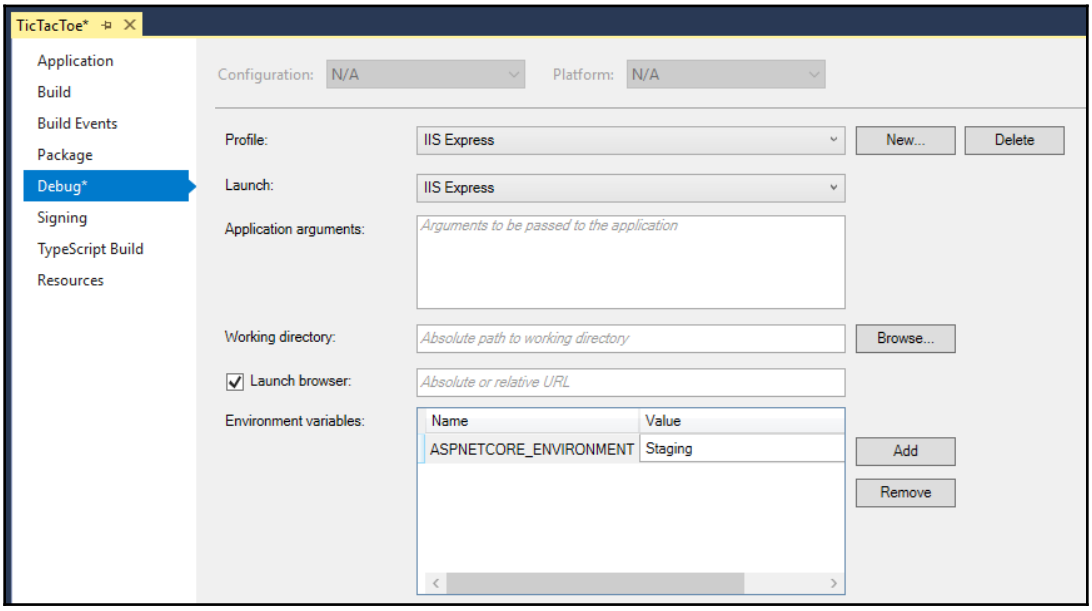


WOW! eBook
www.wowebook.org

8. Right-click on **Update all files** and select **Run**. Now, you can see and understand what the process is doing in more detail:



9. Schedule the process for execution after each build by right-clicking on **Update all files** and selecting **Bindings** | **After build**. A new file called `bundleconfig.json.bindings` will be generated, and, if you remove the `wwwroot/js` folder and rebuild the project, the files will be auto-generated.
10. To see the newly generated files in action, go to the **Debug*** tab in the project settings and set the `ASPNETCORE_ENVIRONMENT` variable to `Staging`. Then, click **Save**:



WOW! eBook
www.wowebook.org

11. Start the application by pressing *F5*, open the developer tools by pressing *F12* in Microsoft Edge, and redo the registration process. You will see that only the bundled and minified `site.min.css` and `site.min.js` files have been loaded and that the load times are faster:

F12 DOM Explorer Console Debugger Network Performance Memory Emulation Experiments

Content type

Name / Path	Protocol	Method	Result / Description	Content type	Received	Time	Initiator / Type	0ms 200ms
CheckEmailConfirmationStatus?email=example@ex... http://localhost:52872/	HTTP	GET	200 OK			31,93 ms	XMLHttpRequest	
GameInvitation?email=example@example.com http://localhost:52872/	HTTP	GET	200 OK	text/html		54,04 ms	document	
bootstrap.min.css https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/	HTTPS	GET	200	text/css	(from cache)	0 s		
site.min.css?v=QBVhELvuZMhWT2PUTfvl_nAuZm8N... http://localhost:52872/css/	HTTP	GET	200 OK	text/css	(from cache)	0 s		
site.min.css http://localhost:52872/css/	HTTP	GET	200 OK	text/css	(from cache)	0 s		
site.min.js http://localhost:52872/js/	HTTP	GET	200 OK	application/java...	(from cache)	0 s		

OK, now that we know how to implement the client-side and benefit from bundling and minification in modern web application development, let's return to the Tic-Tac-Toe game and optimize it even further and add the missing components.

First, we will look at using Web Sockets for real-time communication.

Working with WebSockets for real-time communication scenarios

At the end of the previous section, everything was working fully automated, as expected. However, there is still some room for additional improvements.

As it is, the client-side sends periodical requests to the server-side to see if the email confirmation status has changed. This may lead to a lot of requests to see if there has been a status change or not.

Furthermore, the server-side cannot inform the client-side as soon as an email has been confirmed since it has to wait for a client request to respond to.

In this section, you will learn about the concepts of WebSockets (`https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets`) and how they will allow you to optimize your client-side implementations even more.

WOW! eBook
www.wowebook.org

WebSockets enable persistent two-way communication channels over TCP, which is especially interesting for applications that need to run real-time communication scenarios (chat, stock tickers, games, and more). It just so happens that our example application is a game, which is one of the main application types that largely benefits from working directly with a socket connection.

Note that you could also consider SignalR as an alternative. SignalR provides a better solution for real-time communication scenarios and encapsulates some of the functionalities that are missing from WebSockets that we may have implemented manually.
We will cover SignalR in the next chapter, that is, `Chapter 6`, *Introducing Razor Components and SignalR*.

WebSockets in action

Let's optimize the client-side implementation of the Tic-Tac-Toe application by using WebSockets for real-time communication:

1. Go to the Tic-Tac-Toe `Startup` class in the `Configure` method and add the WebSockets middleware just before the communication middleware and the MVC middleware (remember that the middleware invocation order is important for assuring correct behavior):

```
app.UseWebSockets();
app.UseCommunicationMiddleware();
...
```

2. Update the communication middleware and add two new methods, with the first one being `SendStringAsync`, as follows:

```
private static Task SendStringAsync(WebSocket socket,
     string data, CancellationToken ct =
      default(CancellationToken))
    {
      var buffer = Encoding.UTF8.GetBytes(data);
      var segment = new ArraySegment<byte>(buffer);
      return socket.SendAsync(segment,
WebSocketMessageType.Text,
       true, ct);
    }
```


WOW! eBook
www.wowebook.org

The second one is `ReceiveStringAsync` and is used for WebSockets communication:

```
private static async Task<string> ReceiveStringAsync(
  WebSocket socket, CancellationToken ct =
    default(CancellationToken))
       {
         var buffer = new ArraySegment<byte>(new byte[8192]);
         using (var ms = new MemoryStream())   {
           WebSocketReceiveResult result;
           do
           { ct.ThrowIfCancellationRequested();
             result = await socket.ReceiveAsync(buffer, ct);
             ms.Write(buffer.Array, buffer.Offset, result.Count);
}

           while (!result.EndOfMessage);
           ms.Seek(0, SeekOrigin.Begin);
           if (result.MessageType != WebSocketMessageType.Text)
             throw new Exception("Unexpected message");

           using (var reader = new StreamReader(ms, Encoding.UTF8))
           {  return await reader.ReadToEndAsync();   }
         }
       }
```

3. Update the communication middleware and add a new method called `ProcessEmailConfirmation` for email confirmation processing via WebSockets:

```
public async Task ProcessEmailConfirmation(HttpContext context,
WebSocket currentSocket, CancellationToken ct, string email)

   {
   UserModel user = await _userService.GetUserByEmail(email);
   while (!ct.IsCancellationRequested &&
    !currentSocket.CloseStatus.HasValue &&
           user?.IsEmailConfirmed == false)    {
            if (user.IsEmailConfirmed)
              await SendStringAsync(currentSocket, "OK", ct);
            else
            { user.IsEmailConfirmed = true;
              user.EmailConfirmationDate = DateTime.Now;
              await _userService.UpdateUser(user);
              await SendStringAsync(currentSocket, "OK", ct);   }

            Task.Delay(500).Wait();
            user = await _userService.GetUserByEmail(email);
```


WOW! eBook
www.wowebook.org

```
            }
          }
```

4. Update the `Invoke` method in the communication middleware and add calls to the WebSockets-specific methods from the previous step, while still keeping the standard implementations for browsers that do not support WebSockets:

```
public async Task Invoke(HttpContext context) {
  if (context.WebSockets.IsWebSocketRequest)
    {
      var webSocket = await context.WebSockets.
        AcceptWebSocketAsync();
      var ct = context.RequestAborted;
      var json = await ReceiveStringAsync(webSocket, ct);
      var command = JsonConvert.DeserializeObject<dynamic>(json);

      switch (command.Operation.ToString()) {
         case "CheckEmailConfirmationStatus":
          {await ProcessEmailConfirmation(context, webSocket,
             ct, command.Parameters.ToString());
                break;     }
       }
          }
   else if
(context.Request.Path.Equals("/CheckEmailConfirmationStatus"))
          //... keep the rest of the method as it was
}
```

5. Modify the `scripts1.js` file and add some WebSockets-specific code for opening and working with sockets:

```
var interval;
function EmailConfirmation(email) {
  if (window.WebSocket) {
    alert("Websockets are enabled");
    openSocket(email, "Email");
  }
  else {
    alert("Websockets are not enabled");
    interval = setInterval(() => {
      CheckEmailConfirmationStatus(email);
    }, 5000);
  }
}
```

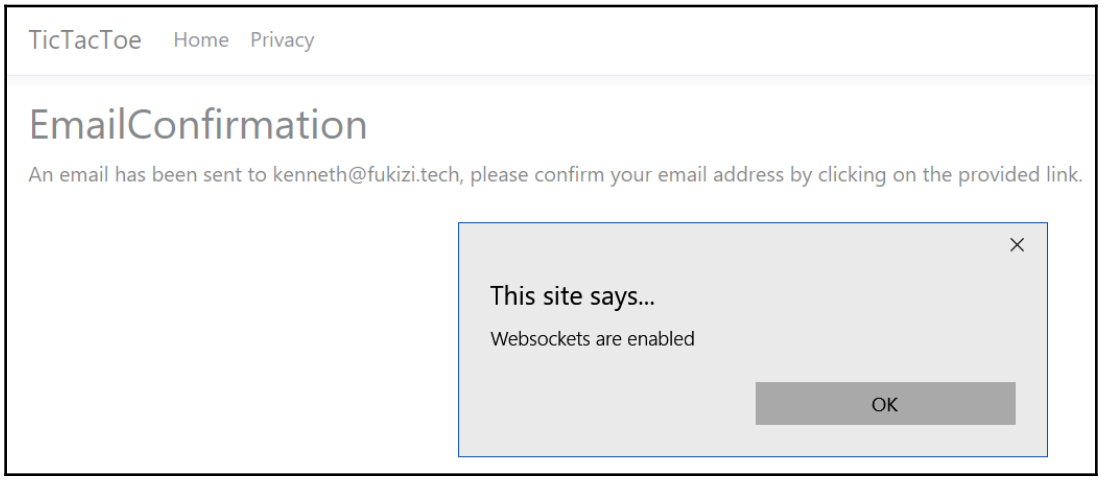

WOW! eBook
www.wowebook.org

6. Modify the `scripts2.js` file but keep the `CheckEmailConfirmationStatus` function the same. Add some WebSockets-specific code for opening and working with sockets and also redirecting the user to the game invitation page if their email has been confirmed:

```
var openSocket = function (parameter, strAction) {
  if (interval !== null) clearInterval(interval);

  var protocol = location.protocol === "https:" ?  "wss:" : "ws:";
  var operation = "";    var wsUri = "";

  if (strAction == "Email") {
     wsUri = protocol + "//" + window.location.host
      + "/CheckEmailConfirmationStatus";
     operation = "CheckEmailConfirmationStatus";         }

  var socket = new WebSocket(wsUri);
     socket.onmessage = function (response) {
       console.log(response);
     if (strAction == "Email" && response.data == "OK") {
            window.location.href = "/GameInvitation?email=" +
             parameter;     }  };

  socket.onopen = function () {
          var json = JSON.stringify({  "Operation": operation,
           "Parameters": parameter   });
          socket.send(json);        };

 socket.onclose = function (event) {   };
};
```


WOW! eBook
www.wowebook.org

7. When you start the application and proceed with the user registration, you will get the necessary information if WebSockets are supported. If they are, you will be redirected to the game invitation page like before, but with the benefit of a much faster processing time:



This concludes our look at client-side development and optimization under ASP.NET Core 3 for the moment. Now, you are going to learn how to further extend and finalize the Tic-Tac-Toe application with additional ASP.NET Core concepts that will help you in your daily work while building multi-lingual, production-ready web applications.

As a web application gets busier, we might want to minimize unnecessary round trips that are requesting data that can be kept for a period of time for retrieval and usage. Let's look at how to do this by introducing session and user cache management.

Taking advantage of session and user cache management

As a web developer, you might know that HTTP is a stateless protocol, which means that, by default, there is not a notion of sessions as such. Each request is handled independently and no values are retained between different requests.

WOW! eBook
www.wowebook.org

Nonetheless, there are different methods for working with data. You can work with query strings, submit form data, or you can use cookies to store data on the client. However, all of those mechanisms are more or less manual and need to be managed by yourself.

If you are an experienced ASP.NET developer, you will be familiar with the concepts of session state and session variables. Those variables are stored on the web server and you can access them during different user requests so that you have a central place to store and receive data. Session state is ideal for storing user data specific to a session, without the need for permanent persistence.

Note that it is good practice to not store any sensitive data in session variables due to security reasons. Users might not close their browsers; thus, session cookies might not be cleared (also, some browsers keep session cookies alive).

Also, a session might not be restricted to a single user, so other users might continue with the same session, which could cause security risks.

ASP.NET Core 3 provides session state and session variables by using a dedicated session middleware. Basically, there are two distinct types of session providers:

- In-memory session providers (locally to a single server)
- Distributed session providers (shared between multiple servers)

In-memory session providers

Let's learn how to activate the in-memory session provider in the Tic-Tac-Toe application for storing the user interface's culture and language:

1. Open the layout page in the `Views\Shared\_Layout.cshtml` file and add a new **User Interface Language Drop-Down** to the main navigation menu. This will be placed after the other menu items. This will allow users to select between English and French:

```
<li class="dropdown">
  <a class="dropdown-toggle" data-toggle="dropdown"
    href="#">Settings<span class="caret"></span></a>
  <ul class="dropdown-menu multi-level">
    <li class="dropdown-submenu">
      <a class="dropdown-toggle" data-toggle="dropdown"
       href="#">Select your language (@ViewBag.Language)
       <span class="caret"></span></a>
      <ul class="dropdown-menu">
```


WOW! eBook
www.wowebook.org

```
                <li @(ViewBag.Language == "EN" ? "active" : "")>
                  <a asp-controller="Home" asp-action="SetCulture"
                   asp-route-culture="EN">English</a></li>
                <li @(ViewBag.Language == "FR" ? "active" : "")>
                  <a asp-controller="Home" asp-action="SetCulture"
                   asp-route-culture="FR">French</a></li>
              </ul>
            </li>
          </ul>
        </li>
```

2. Open `HomeController` and add a new method called `SetCulture`. This will contain the code for storing the user culture settings in a session variable:

```
using Microsoft.AspNetCore.Http;
public IActionResult SetCulture(string culture)
{
  Request.HttpContext.Session.SetString("culture", culture);
  return RedirectToAction("Index");
}
```

3. Update the `Index` method of `HomeController` in order to retrieve the culture from the culture session variable:

```
public IActionResult Index()
{
  var culture =
    Request.HttpContext.Session.GetString("culture");
  ViewBag.Language = culture;
  return View();
}
```

4. Go to the `wwwroot/css/site.css` file and add some new CSS classes for a more modern look for the **User Interface Language Drop-Down**, firstly for relative positions, and then for different browsers and hovering:

```
.dropdown-submenu {
          position: relative;
        }

        .dropdown-submenu > .dropdown-menu {
          top: 0;
          left: 100%;
          margin-top: -6px;
          margin-left: -1px;
          -webkit-border-radius: 0 6px 6px 6px;
          -moz-border-radius: 0 6px 6px;
          border-radius: 0 6px 6px 6px;
```


WOW! eBook
www.wowebook.org

```
}

.dropdown-submenu:hover > .dropdown-menu {
  display: block;
}
```

Do the same in `site.css` by adding the following styling:

```
.dropdown-submenu > a:after {
  display: block;
  content: " ";
  float: right;
  width: 0;
  height: 0;
  border-color: transparent;
  border-style: solid;
  border-width: 5px 0 5px 5px;
  border-left-color: #ccc;
  margin-top: 5px;
  margin-right: -10px;
}

.dropdown-submenu:hover > a:after {
  border-left-color: #fff;
}
```

Finally, add the following snippet:

```
.dropdown-submenu.pull-left {
  float: none;
}

.dropdown-submenu.pull-left > .dropdown-menu {
  left: -100%;
  margin-left: 10px;
  -webkit-border-radius: 6px 0 6px 6px;
  -moz-border-radius: 6px 0 6px 6px;
  border-radius: 6px 0 6px 6px;
}
```

5. Add the built-in session middleware of ASP.NET Core 3 to the `ConfigureServices` method of the `Startup` class:

```
services.AddSession(o =>
{
  o.IdleTimeout = TimeSpan.FromMinutes(30);
});
```


WOW! eBook
www.wowebook.org

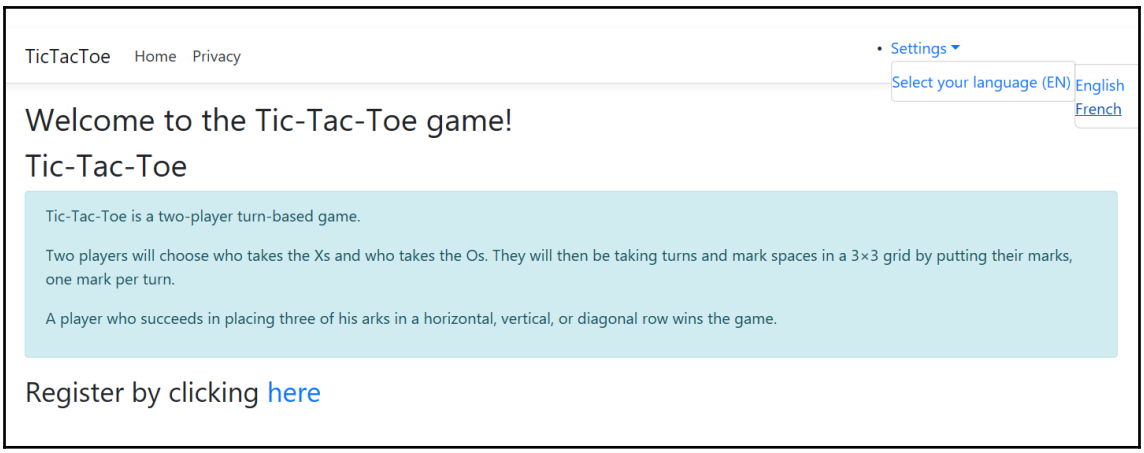
6. Activate the session middleware in the `Configure` method of the `Startup` class by adding it just after the static files middleware:

```
app.UseStaticFiles();
app.UseSession();
```

7. Update the `Index` method in `GameInvitationController` by setting the email session variable, as follows:

```
[HttpGet]
public async Task<IActionResult> Index(string email)
{
    var gameInvitationModel = new GameInvitationModel {
      InvitedBy = email };
    HttpContext.Session.SetString("email", email);
    return View(gameInvitationModel);
}
```

8. Start the application by pressing *F5*. You should see the new **User Interface Language Drop-Down** with the options to select between English and French:



Here, we have used an in-memory session provider, but there is a different type of session provider that works well in other scenarios and is called a distributed session provider. We will look at this in the next section.

WOW! eBook
www.wowebook.org

Distributed session providers

So far, you have learned how to activate and use session state. However, most of the time, you will have multiple web servers, not just one, especially in today's cloud environments. So, how do you store session state out of memory in a distributed cache?

Well, that's easy – you just have to register additional services within the `Startup` class. These additional services will provide this functionality. Here are some examples:

- **Distributed Memory Cache**:

```
services.AddDistributedMemoryCache();
```

- **Distributed SQL Server Cache**:

```
services.AddDistributedSqlServerCache(o =>
{
  o.ConnectionString =
_configuration["DatabaseConnection"];
  o.SchemaName = "dbo";
  o.TableName = "sessions";
});
```

- **Distributed Redis Cache**:

```
services.AddDistributedRedisCache(o =>
{
  o.Configuration =
_configuration["CacheRedis:Connection"];
  o.InstanceName = _configuration
    ["CacheRedis:InstanceName"];
});
```

We have added a new **User Interface Language Drop-Down** in this section, but you haven't learned how to handle multiple languages within your applications yet. There's no time to lose; let's learn how to do this and use the drop-down and session variable for changing the user interface language on the fly.

WOW! eBook
www.wowebook.org

Applying globalization and localization for multi-lingual user interfaces

Sometimes, your applications achieve success, sometimes even very considerable success, and so you want to provide them internationally to a wider audience and deploy them at a larger scale. However, you can't do this easily because you haven't thought of localizing your applications from the beginning, and now you have to modify your already-running application with the risk of regressions and destabilizations.

Don't fall into this trap! Think about your target audience and future deployment strategy from the start!

Localizing your applications should be considered from the beginning of your projects, especially since it is very easy and straightforward to do when using the ASP.NET Core 3 Framework. It provides existing services and middleware for this purpose.

Building applications that support different languages and cultures for display, input, and output is called **globalization**, whereas adapting a globalized application to a specific culture is called localization.

There are three different methods for localizing ASP.NET Core 3 web applications:

- The string localizer
- The view localizer
- Localizing Data Annotations

Let's take a look at these concepts in more detail.

Globalization and localization concepts

In this section, you will learn about the concepts of globalization and localization and how they will allow you to further optimize your websites for internationalization.

For additional information on globalization and localization, please visit `https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization`.

WOW! eBook
www.wowebook.org

So, how do you get started? Well, first of all, let's look at how to make the Tic-Tac-Toe application localizable by using the String Localizer:

1. Go to the `Services` folder and add a new service called `CultureProviderResolverService`. This will retrieve the culture set by looking at the `Culture` query string, the `Culture` cookie, and the `Culture` session variable (created in the previous section of this chapter).
2. Implement `CultureProviderResolverService` by inheriting it from `RequestCultureProvider` and overriding its specific methods:

```
public class CultureProviderResolverService :
    RequestCultureProvider
{
  private static readonly char[] _cookieSeparator =
    new[] {'|' };
  private static readonly string _culturePrefix = "c=";
  private static readonly string _uiCulturePrefix = "uic=";
   //...
}
```

3. Add the `DetermineProviderCultureResult` method to the `CultureProviderResolverService` class:

```
public override async Task<ProviderCultureResult>
           DetermineProviderCultureResult(HttpContext httpContext)
          {
            if (GetCultureFromQueryString(httpContext,
             out string culture))
            return new ProviderCultureResult(culture, culture);

            else if (GetCultureFromCookie(httpContext, out
culture))
            return new ProviderCultureResult(culture, culture);

            else if (GetCultureFromSession(httpContext, out
            culture))
            return new ProviderCultureResult(culture, culture);

            return await NullProviderCultureResult;
          }
```


WOW! eBook
www.wowebook.org

4. Add the following method, which will allow us to get `Culture` from a query string:

```
private bool GetCultureFromQueryString(   HttpContext httpContext,
out string culture)
          {
            if (httpContext == null)
            {
              throw new ArgumentNullException(nameof(httpContext));
            }

            var request = httpContext.Request;
            if (!request.QueryString.HasValue)
            {
              culture = null;
              return false;
            }

            culture = request.Query["culture"];
            return true;
          }
```

5. Add the following method, which will allow us to get `Culture` from cookies:

```
private bool GetCultureFromCookie(HttpContext httpContext, out
    string culture)
          {
            if (httpContext == null)
            {
              throw new ArgumentNullException(nameof(httpContext));
            }

            var cookie = httpContext.Request.Cookies["culture"];
            if (string.IsNullOrEmpty(cookie))
            {
              culture = null;
              return false;
            }

            culture = ParseCookieValue(cookie);
            return !string.IsNullOrEmpty(culture);
          }
```


WOW! eBook
www.wowebook.org

6. Here's the method that we will use to parse a cookie value:

```
public static string ParseCookieValue(string value)
{
  if (string.IsNullOrWhiteSpace(value))   return null;
  var parts = value.Split(_cookieSeparator,
    StringSplitOptions.RemoveEmptyEntries);
  if (parts.Length != 2)  return null;

  var potentialCultureName = parts[0];
  var potentialUICultureName = parts[1];

  if (!potentialCultureName.StartsWith(_culturePrefix) ||
              !potentialUICultureName.StartsWith(_uiCulturePrefix))
return null;

  var cultureName = potentialCultureName.
    Substring(_culturePrefix.Length);
  var uiCultureName = potentialUICultureName.Substring
    (_uiCulturePrefix.Length);
  if (cultureName == null && uiCultureName == null) return null;
  if (cultureName != null && uiCultureName == null) uiCultureName =
    cultureName;

  if (cultureName == null && uiCultureName != null)  cultureName =
uiCultureName;
 return cultureName;
}
```

7. Now, add the following method, which allows us to get `Culture` from the session:

```
private bool GetCultureFromSession(HttpContext httpContext,
      out string culture)
{
  culture = httpContext.Session.GetString("culture");
  return !string.IsNullOrEmpty(culture);
}
```

8. Add the localization service at the top of the `ConfigureServices` method in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddLocalization(options => options.
    ResourcesPath = "Localization");
  //...
}
```


WOW! eBook
www.wowebook.org

9. Add the localization middleware to the `Configure` method in the `Startup` class and define the supported cultures.

   Note that the order of adding middleware is important, as you have already seen. You have to add the localization middleware just before the MVC middleware:

```
...
var supportedCultures =
  CultureInfo.GetCultures(CultureTypes.AllCultures);
var localizationOptions = new RequestLocalizationOptions
{
  DefaultRequestCulture = new RequestCulture("en-US"),
  SupportedCultures = supportedCultures,
  SupportedUICultures = supportedCultures
};

localizationOptions.RequestCultureProviders.Clear();
localizationOptions.RequestCultureProviders.Add(new
 CultureProviderResolverService());

app.UseRequestLocalization(localizationOptions);

app.UseMvc(...);
```

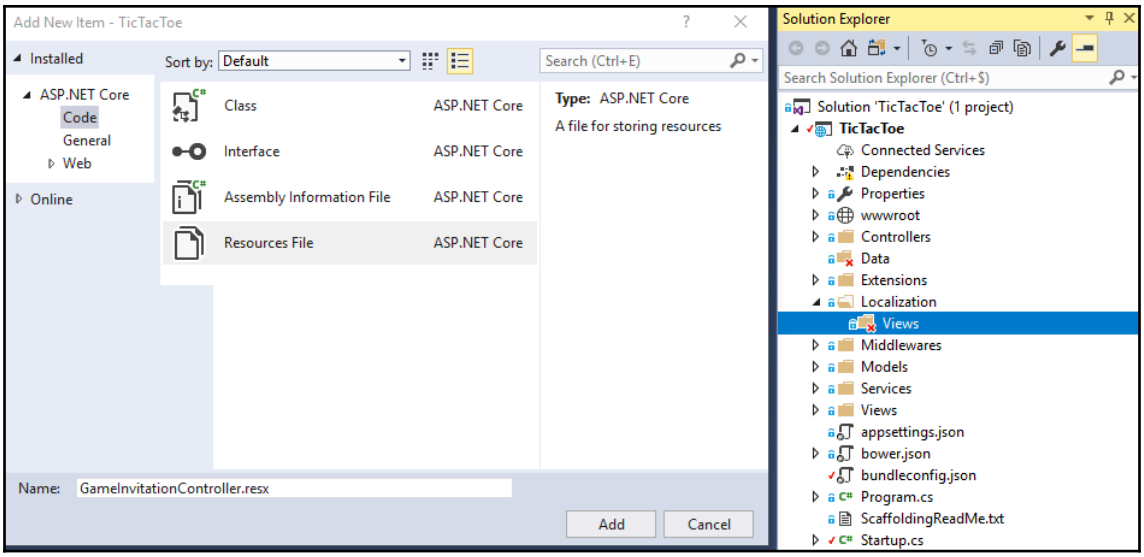
Note that you can use different methods to change the culture of your applications:

- **Query strings**: Provide the culture in the URI
- **Cookies**: Store the culture in a cookie
- **Browser**: Browser page language settings
- **Custom**: Implement your own provider (shown in this example)

10. In the **Solution Explorer**, add a new folder called `Localization` (it will be used to store the resource files) and create a subfolder called `Controllers`. Then, within this folder, add a new resource file called `GameInvitationController.resx`:

    Note that you can put your resource files either into subfolders (for example, `Controllers`, `Views`, and so on) or directly name your files accordingly (for example, `Controllers.GameInvitationController.resx`, `Views.Home.Index.resx`, and so on). However, we advise that you use the folder approach for clarity, readability, and better organization of your files.

WOW! eBook
www.wowebook.org



If you see errors while using your resource files with .NET Core, right-click on each file and select **Properties**. Then, check each file to ensure that the **Build Action** is set to **Content** instead of **Embedded Resource**. These are bugs that should have been fixed by the final release, but if they haven't you can use this handy workaround to make everything work as expected.

11. Open the `GameInvitationController.resx` resource file and add a new `GameInvitationConfirmationMessage` in English:

	Name	Value
▶	GameInvitationConfirmationMessage	You have invited {0} for the next game.
*		

12. In the same `Controllers` folder, add a new resource file for the French translations called `GameInvitationController.fr.resx`:

	Name	Value
▶	GameInvitationConfirmationMessage	Vous avez invité {0} pour la prochaine partie.
*		

WOW! eBook
www.wowebook.org

13. Go to `GameInvitationController`, add `stringLocalizer`, and update the constructor implementation:

```
        private IStringLocalizer<GameInvitationController>
         _stringLocalizer;
        private IUserService _userService;
        public GameInvitationController(IUserService userService,
         IStringLocalizer<GameInvitationController>
stringLocalizer)
        {
          _userService = userService;
          _stringLocalizer = stringLocalizer;
        }
```

14. Add a new `Index` method to `GameInvitationController`. This will return a localized message, depending on the application locale settings:

```
        [HttpPost]
        public IActionResult Index(
         GameInvitationModel gameInvitationModel)
        {
          return Content(_stringLocalizer[
          "GameInvitationConfirmationMessage",
           gameInvitationModel.EmailTo]);
        }
```

15. Start the application in English (the default culture) and register a new user until you get the following text message, which should be in English:

WOW! eBook
www.wowebook.org

16. Change the application language to French by using the **User Interface Language Drop-Down**. Then, register a new user until you get the following text message, which should now be in French, as shown in the following screenshot:

In this section, you have learned how to localize any type of string within your applications, which can be useful for some of your specific application use cases. However, this is not the recommended approach when working with views.

Using the view localizer

The ASP.NET Core 3 Framework provides some powerful features for localizing views. You are going to use the view localizer approach in the following example:

1. Update the `ConfigureServices` method in the `Startup` class and add the view localization service to the MVC service declaration:

```
services.AddMvc().AddViewLocalization(
  LanguageViewLocationExpanderFormat.Suffix,
  options => options.ResourcesPath = "Localization");
```

2. Modify the `Views/ViewImports.cshtml` file and add the view localizer functionalities so that they will be available for all the views:

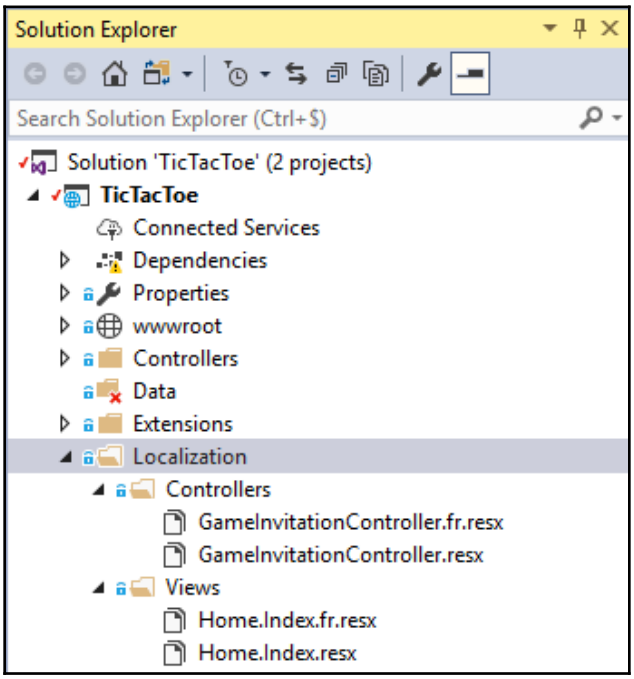
```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
```

3. Open the home page view and add a new title, which is going to be localized further, as follows:

```
<h2>@Localizer["Title"]</h2>
```


WOW! eBook
www.wowebook.org

4. In the Solution Explorer, go to the `Localization` folder and create a subfolder called `Views`. Then, add two new resource files called `Home.Index.resx` and `Home.Index.fr.resx` to this folder:



5. Open the `Home.Index.resx` file and add an entry for the English title:

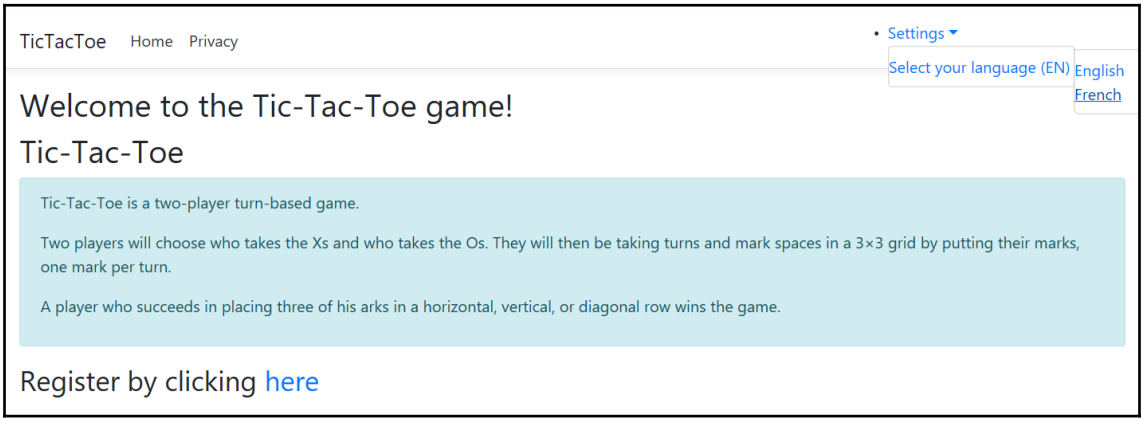
Name	Value
Title	Welcome to the Tic-Tac-Toe Game!

6. Open the `Home.Index.fr.resx` file and add an entry for the French title:

Name	Value
Title	Bienvenue sur le jeu du Morpion!

WOW! eBook
www.wowebook.org

7. Start the application and set the user interface language drop-down to English:



8. Change the application language to French using the **User Interface Language Drop-Down**. The title should now be displayed in French:

In this section, you've seen how to easily localize your views, but how do you localize forms that are using Data Annotations within your views? Let's look at this in more detail; you will be surprised at what the ASP.NET Core 3 Framework has to offer!

WOW! eBook
www.wowebook.org

Localizing Data Annotations

We are going to completely localize the user registration form in the following examples:

1. In the Solution Explorer, go to the `Localization/Views` folder and add two new resource files called `UserRegistration.Index.resx` and `UserRegistration.Index.fr.resx`.
2. Open the `UserRegistration.Index.resx` file and add a `Title` and a `SubTitle` element with English translations:

Name	Value
Title	User Registration
SubTitle	User Record

3. Open the `UserRegistration.Index.fr.resx` file and add a `Title` and a `SubTitle` element with French translations:

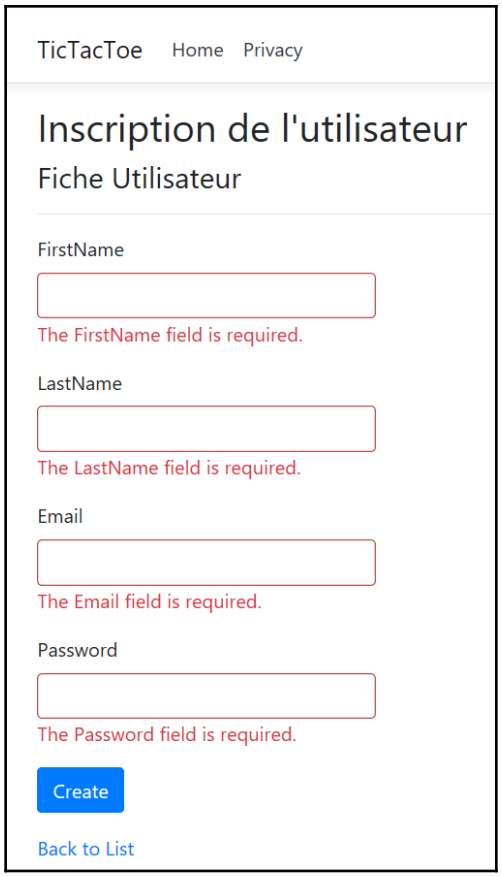
Name	Value
Title	Inscription de l'utilisateur
SubTitle	Fiche Utilisateur

4. Update the **User Registration Index View** so that it uses the **View Localizer**:

```
@model TicTacToe.Models.UserModel
@{
   ViewData["Title"] = Localizer["Title"];
}
<h2>@ViewData["Title"]</h2>
<h4>@Localizer["SubTitle"]</h4>
<hr />
<div class="row">
...
```

5. Start the application, set the language to French using the **User Interface Language Drop-Down**, and go to the user registration page. The titles should be displayed in French. Click on **Create** without entering anything in the input fields and see what happens:

WOW! eBook
www.wowebook.org



Something is missing here. You have added localization for the page title, as well as the subtitle of the user registration page, but we are still missing some localizations for the form. But what are we missing?

You should have seen for yourself that the error messages haven't been localized and translated yet. We are using the Data Annotation framework for error handling and form validation, so how do we localize Data Annotation validation error messages? This is what we are going look at now:

1. Add the Data Annotation localization service to the MVC service declaration in the `ConfigureServices` method of the `Startup` class:

```
services.AddMvc().AddViewLocalization(
  LanguageViewLocationExpanderFormat.Suffix, options =>
  options.ResourcesPath = "Localization")
  .AddDataAnnotationsLocalization();
```


WOW! eBook
www.wowebook.org

2. Go to the `Localization` folder and create a subfolder called `Models`. Then, add two new resource files called `UserModel.resx` and `UserModel.fr.resx`.
3. Update the `UserModel.resx` file with English translations:

Name	Value
Email	E-Mail
EmailRequired	The e-mail is required.
FirstName	First Name
FirstNameRequired	The first name is required.
LastName	Last Name
LastNameRequired	The last name is required.
Password	Password
PasswordRequired	The password is required.

4. Next, update the `UserModel.fr.resx` file with French translations:

Name	Value
Email	E-Mail
EmailRequired	L'e-mail est obligatoire.
FirstName	Prénom
FirstNameRequired	Le prénom est obligatoire.
LastName	Nom
LastNameRequired	Le nom est obligatoire.
Password	Mot de passe
PasswordRequired	Le mot de passe est obligatoire.

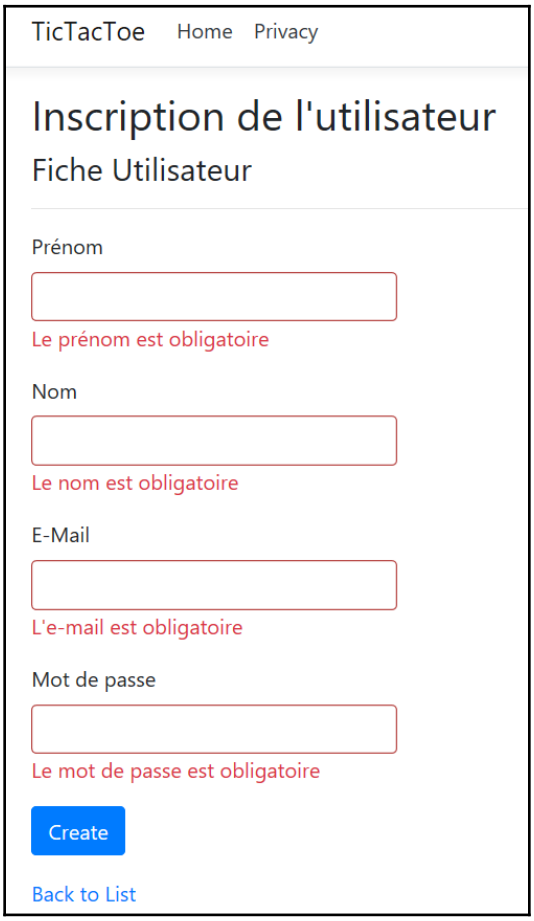
5. Go to the **Models** folder and update the `UserModel` implementation for the `FirstName`, `LastName`, `Email`, and `Password` fields so that you can use the preceding resource files:

```
...
 [Display(Name = "FirstName")]
 [Required(ErrorMessage = "FirstNameRequired")]
 public string FirstName { get; set; }

 [Display(Name = "LastName")]
 [Required(ErrorMessage = "LastNameRequired")]
 public string LastName { get; set; }
```


WOW! eBook
www.wowebook.org

```
    [Display(Name = "Email")]
    [Required(ErrorMessage = "EmailRequired"),
      DataType(DataType.EmailAddress)]
    [EmailAddress]
    public string Email { get; set; }

    [Display(Name = "Password")]
    [Required(ErrorMessage = "PasswordRequired"),
      DataType(DataType.Password)]
    public string Password { get; set; }
...
```

6. Rebuild the solution and start the application. You will see that the whole user registration page, including the error messages, is completely translated when we change the user interface language to French:



WOW! eBook
www.wowebook.org

In this section, you have learned how to localize strings, views, and even error messages using Data Annotations. For that, you have used the built-in features of ASP.NET Core 3 since they contain everything for developing multi-lingual localizable web applications. The next section is going to give you some insights into how to configure your applications and services.

Configuring your applications and services

In the previous sections, you have advanced by adding missing components to the user registration process and even localizing parts of the Tic-Tac-Toe application. However, you have always simulated the confirmation email by setting the user confirmation programmatically in code. In this section, we will modify this part so that we really do send emails to newly registered users and make everything fully configurable.

Adding an email service

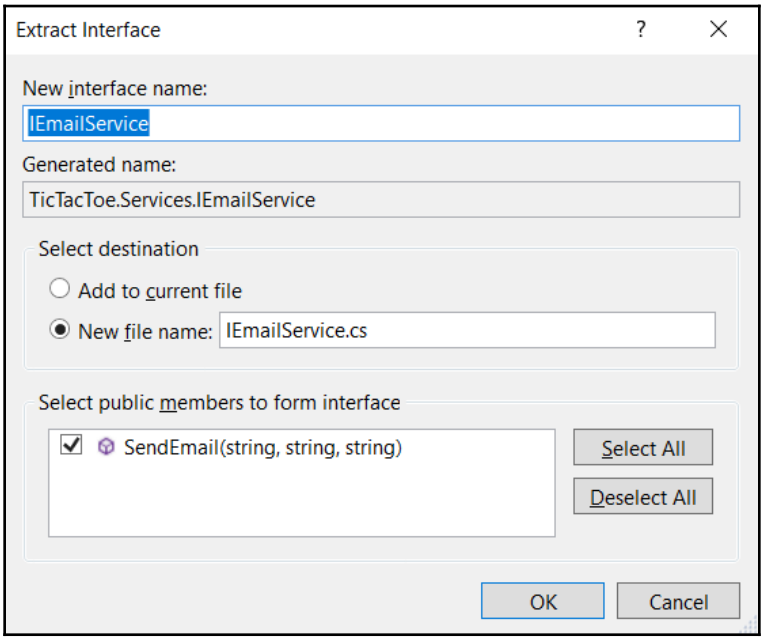
First, you are going to add a new email service, which will be used to send emails to users who have freshly registered on the website. Let's get started:

1. Within the `Services` folder, add a new service called `EmailService` and implement a default `SendEmail` method, which we will update later:

```
public class EmailService
{
  public Task SendEmail(string emailTo, string subject,
   string message)
  {
    return Task.CompletedTask;
  }
}
```


WOW! eBook
www.wowebook.org

2. Extract the `IEmailService` interface:



3. Add the new email service to the `ConfigureServices` method of the `Startup` class (we want a single application instance, so add it as a singleton):

```
services.AddSingleton<IEmailService, EmailService>();
```

4. Update `UserRegistrationController` so that it is able to access `EmailService`, which we created in the previous step:

```
readonly IUserService _userService;
readonly IEmailService _emailService;
public UserRegistrationController(IUserService userService,
 IEmailService emailService)
{
  _userService = userService;
  _emailService = emailService;
}
```


WOW! eBook
www.wowebook.org

5. Update the `EmailConfirmation` method in `UserRegistrationController` so that you can call the `SendEmail` method of `EmailService` by inserting the following code between `var user=await _userService.GetUserByEmail(email);` and the `user?.IsEmailConfirmed` conditional statement check:

```
var user = await _userService.GetUserByEmail(email);
var urlAction = new UrlActionContext
{
  Action = "ConfirmEmail",
  Controller = "UserRegistration",
  Values = new { email },
  Protocol = Request.Scheme,
  Host = Request.Host.ToString()
};

var message = $"Thank you for your registration on
   our website, please click here to confirm your
   email " +   $"
  {Url.Action(urlAction)}";

try
{
  _emailService.SendEmail(email,
   "Tic-Tac-Toe Email Confirmation", message).Wait();
}
catch (Exception e)    {    }
```

Great – you have an email service now, but you aren't done yet. You need to be able to configure the service so that you can set environment-specific parameters (SMTP server name, port, SSL, and more) and then send the emails.

Configuring the email service

Nearly all of the services you create in the future will have some kind of configuration, which should be configurable from the outside of your code.

ASP.NET Core 3 has a built-in Configuration API for this purpose. It provides various functionalities for reading configuration data from multiple sources during application runtime. *Name-value* pairs, which can be grouped into multi-level hierarchies, are used for configuration data persistence. Furthermore, the configuration data can be automatically deserialized into **Plain Old CLR Objects** (**POCO**), which contain private members and properties.

WOW! eBook
www.wowebook.org

The following configuration sources are supported by ASP.NET Core 3:

- Configuration files (JSON, XML, and even classic INI files)
- Environment variables
- Command-line arguments
- In-memory .NET objects
- Encrypted user stores
- Azure Key Vault
- Custom providers

For more information about the Configuration API, please visit `https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration?tabs=basicconfiguration`.

Let's learn how to make the email service quickly configurable by using the ASP.NET Core 3 Configuration API with a JSON configuration file:

1. Add a new `appsettings.json` configuration file to the project and add the following custom section. This will be used to configure the email service:

```
"Email": {
  "MailType": "SMTP",
  "MailServer": "localhost",
  "MailPort": 25,
  "UseSSL": false,
  "UserId": "",
  "Password": "",
  "RemoteServerAPI": "",
  "RemoteServerKey": ""
}
```

2. In the Solution Explorer, create a new folder called `Options` at the root of the project. Add a new POCO class called `EmailServiceOptions` to this folder and implement some private members, as well as public properties, for the options we saw previously:

```
public class EmailServiceOptions
{
  private string MailType { get; set; }
  private string MailServer { get; set; }
  private string MailPort { get; set; }
  private string UseSSL { get; set; }
  private string UserId { get; set; }
```


WOW! eBook
www.wowebook.org

```
    private string Password { get; set; }
    private string RemoteServerAPI { get; set; }
    private string RemoteServerKey { get; set; }

    public EmailServiceOptions()  {  }

    public EmailServiceOptions(string mailType, string mailServer,
      string mailPort, string useSSL,
             string userId, string password, string
              remoteServerAPI,string remoteServerKey)  {
              MailType = mailType;
              MailServer = mailServer;
              MailPort = mailPort;
              UseSSL = useSSL;
              UserId = userId;
              Password = password;
              RemoteServerAPI = remoteServerAPI;
              RemoteServerKey = remoteServerKey;        }
}
```

3. Update the `EmailService` implementation, add `EmailServiceOptions`, and add a parameterized constructor to the class:

```
private EmailServiceOptions _emailServiceOptions;
public EmailService(IOptions<EmailServiceOptions>
 emailServiceOptions)
{
  _emailServiceOptions = emailServiceOptions.Value;
}
```

4. Add a new constructor to the `Startup` class so that you can configure your email service:

```
public IConfiguration _configuration { get; }
public Startup(IConfiguration configuration)
{
  _configuration = configuration;
}
```

5. Update the `ConfigureServices` method of the `Startup` class:

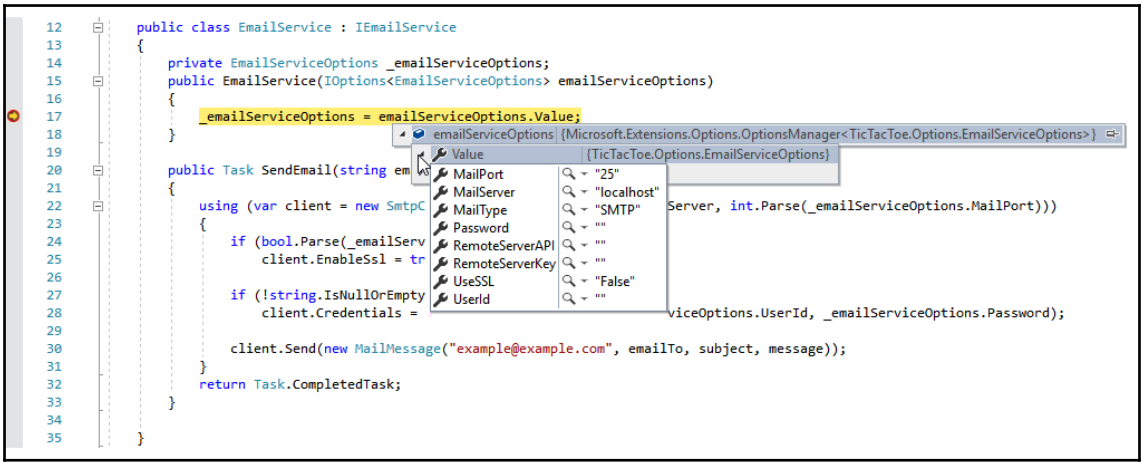
```
services.Configure<EmailServiceOptions>
    (_configuration.GetSection("Email"));
services.AddSingleton<IEmailService, EmailService>();
```


WOW! eBook
www.wowebook.org

6. Update the `SendEmail` method in `EmailService`. Use the email service options to retrieve the settings from the configuration file, as shown here:

```
public Task SendEmail(string emailTo, string
 subject, string message)
{
  using (var client =
    new SmtpClient(_emailServiceOptions.MailServer,
    int.Parse(_emailServiceOptions.MailPort)))
  {
    if (bool.Parse(_emailServiceOptions.UseSSL) ==
     true)client.EnableSsl = true;

    if (!string.IsNullOrEmpty(_emailServiceOptions.UserId))
      client.Credentials =
        new NetworkCredential(_emailServiceOptions.UserId,
         _emailServiceOptions.Password);

      client.Send(new MailMessage("example@example.com",
        emailTo, subject, message));
  }
  return Task.CompletedTask;
}
```

7. Put a breakpoint into the `EmailService` constructor and start the application in debug mode by pressing *F5*. Now, verify that the email service options values have been retrieved correctly from the configuration file. If you have an SMTP server, you can also verify that the email has really been sent:



WOW! eBook
www.wowebook.org

In this section, you have learned how to configure your applications and services by using the built-in Configuration API of ASP.NET Core 3, which allows you to write less code and be much more productive, all while providing a far more elegant and more maintainable solution.

Another feature of ASP.NET Core 3 that helps us have maintainable code is its intrinsic **dependency injection** (**DI**) capabilities. Among other advantages, DI ensures that we don't have too much coupling between classes. We'll have a look at DI in the context of ASP.NET Core 3 in the next section.

Implementing advanced dependency injection concepts

In the previous chapter, you saw how DI works and how to use the constructor injection method. However, if you need to inject many instances during runtime, this method can be quite cumbersome and can make it complicated to understand and maintain your code.

Therefore, you can use a more advanced technique of DI called **method injection**. This allows you to access instances directly from within your code.

Method injection

In the following example, you are going to add a new service for handling game invitations and updating the Tic-Tac-Toe application. This facilitates email communication, which is used for contacting other users to join a game, while using method injection:

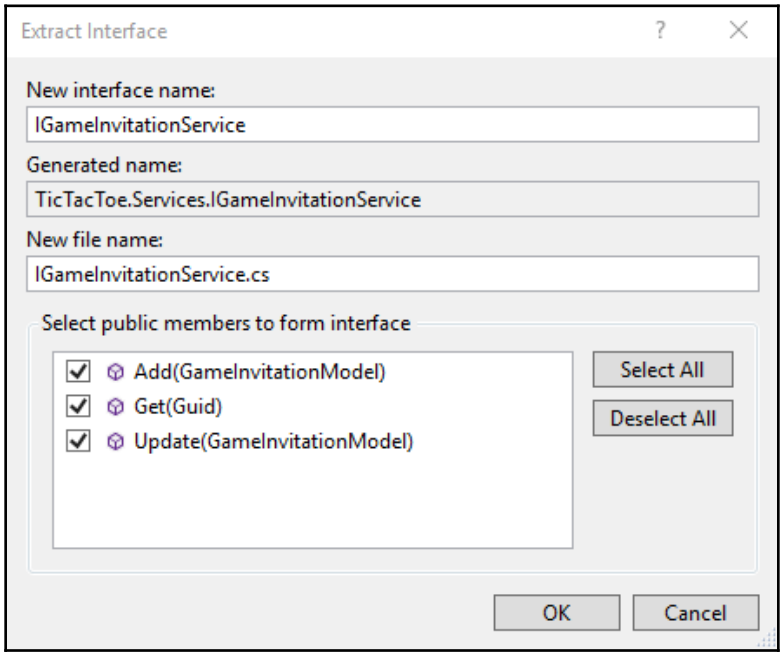
1. Add a new service called `GameInvitationService` in the `Services` folder for managing game invitations (adding, updating, removing, and more):

```
public class GameInvitationService
{
  private static ConcurrentBag<GameInvitationModel>
    _gameInvitations;
  public GameInvitationService(){ _gameInvitations = new
ConcurrentBag<GameInvitationModel>();}

  public Task<GameInvitationModel> Add(GameInvitationModel
    gameInvitationModel)
  {  gameInvitationModel.Id = Guid.NewGuid();
    _gameInvitations.Add(gameInvitationModel);
    return Task.FromResult(gameInvitationModel); }
```


WOW! eBook
www.wowebook.org

```
    public Task Update(GameInvitationModel gameInvitationModel)
    { _gameInvitations = new ConcurrentBag<GameInvitationModel>
          (_gameInvitations.Where(x => x.Id !=
gameInvitationModel.Id))
              {  gameInvitationModel  };
      return Task.CompletedTask;  }

    public Task<GameInvitationModel> Get(Guid id)
    {  return Task.FromResult(_gameInvitations.FirstOrDefault(x =>
      x.Id == id)); }
}
```

2. Extract the `IGameInvitationService` interface:



3. Add the new game invitation service to the `ConfigureServices` method of the `Startup` class (we want a single application instance, so add it as a singleton):

```
services.AddSingleton<IGameInvitationService,
 GameInvitationService>();
```


WOW! eBook
www.wowebook.org

4. Update the `Index` method in `GameInvitationController` and inject an instance of the game invitation service via method injection using the `RequestServices` provider:

```
public IActionResult Index(GameInvitationModel gameInvitationModel,
[FromServices]IEmailService emailService)
{
 var gameInvitationService =
Request.HttpContext.RequestServices.GetService
<IGameInvitationService>();
  if (ModelState.IsValid) {
     emailService.SendEmail(gameInvitationModel.EmailTo,
     _stringLocalizer["Invitation for playing a Tic-Tac-Toe game"],
     _stringLocalizer[$"Hello, you have been invited to play the
    Tic-Tac-Toe game by {0}. For joining the game, please
    click here {1}", gameInvitationModel.InvitedBy,
    Url.Action("GameInvitationConfirmation", GameInvitation",
     new { gameInvitationModel.InvitedBy,
          gameInvitationModel.EmailTo }, Request.Scheme,
            Request.Host.ToString())]);
  var invitation =  gameInvitationService.Add
    (gameInvitationModel).Result;
  return RedirectToAction("GameInvitationConfirmation",
     new { id = invitation.Id });   }
  return View(gameInvitationModel);
}
```

Don't forget to add the following using statement at the beginning of the class: `using Microsoft.Extensions.DependencyInjection;`, If you don't, the `.GetService<IGameInvitationService>();` method can't be used and you will get build errors.

5. Add a new method called `GameInvitationConfirmation` to `GameInvitationController`:

```
        [HttpGet]
        public IActionResult GameInvitationConfirmation(Guid id,
         [FromServices]IGameInvitationService
gameInvitationService)
        {
          var gameInvitation =
gameInvitationService.Get(id).Result;
          return View(gameInvitation);
        }
```


WOW! eBook
www.wowebook.org

6. Create a new view for the `GameInvitationConfirmation` method you added previously. This will display a waiting message to the user:

```
@model TicTacToe.Models.GameInvitationModel
@{
   ViewData["Title"] = "GameInvitationConfirmation";
   Layout = "~/Views/Shared/_Layout.cshtml";
}
<h1>@Localizer["You have invited {0} to play
 a Tic-Tac-Toe game
 with you, please wait until the user is connected",
 Model.EmailTo]</h1>
@section Scripts{
  <script>
    $(document).ready(function () {
      GameInvitationConfirmation('@Model.Id');
    });
  </script>
}
```

7. Add a new method called `GameInvitationConfirmation` to the `scripts1.js` file. You can use the same basic structure we used for the existing `EmailConfirmation` method:

```
function GameInvitationConfirmation(id) {
  if (window.WebSocket) {
    alert("Websockets are enabled");
    openSocket(id, "GameInvitation");
  }
  else {
    alert("Websockets are not enabled");
    interval = setInterval(() => {
      CheckGameInvitationConfirmationStatus(id);
    }, 5000);
  }
}
```

8. Add a method called `CheckGameInvitationConfirmationStatus` to the `scripts2.js` file. You can use the same basic structure we used for the existing `CheckEmailConfirmationStatus` method:

```
function CheckGameInvitationConfirmationStatus(id) {
  $.get("/GameInvitationConfirmation?id=" + id,
  function (data) {
    if (data.result === "OK") {
      if (interval !== null)
        clearInterval(interval);
```


WOW! eBook
www.wowebook.org

```
        window.location.href = "/GameSession/Index/" + id;
      }
    });
  }
```

9. Update the `openSocket` method in the `scripts2.js` file and add the specific game invitation case:

```
...
if (strAction == "Email") {
  wsUri = protocol + "//" + window.location.host +
"/CheckEmailConfirmationStatus";
  operation = "CheckEmailConfirmationStatus";
}
else if (strAction == "GameInvitation") {
  wsUri = protocol + "//" + window.location.host +
"/GameInvitationConfirmation";
  operation = "CheckGameInvitationConfirmationStatus";
}

var socket = new WebSocket(wsUri);
socket.onmessage = function (response) { console.log(response);
  if (strAction == "Email" && response.data == "OK") {
    window.location.href = "/GameInvitation?email=" + parameter;
  }else if (strAction == "GameInvitation") {
    var data = $.parseJSON(response.data);

  if (data.Result == "OK") window.location.href =
"/GameSession/Index/" + data.Id; }  };
 ...
```

10. Add a new method called `ProcessGameInvitationConfirmation` in the communication middleware. This will process game invitation requests without using WebSockets for browsers that don't support this feature:

```
private async Task ProcessGameInvitationConfirmation(HttpContext
context)
{
  var id = context.Request.Query["id"];
  if (string.IsNullOrEmpty(id))await context.
   Response.WriteAsync("BadRequest:Id is required");

  var gameInvitationService =  context.RequestServices.GetService
    <IGameInvitationService>();
  var gameInvitationModel =  await
    gameInvitationService.Get(Guid.Parse(id));

  if (gameInvitationModel.IsConfirmed) await
```


WOW! eBook
www.wowebook.org

```
    context.Response.WriteAsync(
             JsonConvert.SerializeObject(new
    {
      Result = "OK",
      Email = gameInvitationModel.InvitedBy,
       gameInvitationModel.EmailTo
    }));
  else {
         await context.Response.WriteAsync(
          "WaitGameInvitationConfirmation");
       }
 }
```

Don't forget to add the following using statement at the beginning of the class:
`using Microsoft.Extensions.DependencyInjection;`.

11. Add a new method called `ProcessGameInvitationConfirmation` with additional parameters to the communication middleware. This will process game invitation requests while using WebSockets for the browsers that support this:

```
private async Task ProcessGameInvitationConfirmation(HttpContext
context,
       WebSocket webSocket, CancellationToken ct,
       string parameters)
{
  var gameInvitationService = context.RequestServices.GetService
    <IGameInvitationService>();
  var id = Guid.Parse(parameters);
  var gameInvitationModel = await gameInvitationService.Get(id);
  while (!ct.IsCancellationRequested && !webSocket.
    CloseStatus.HasValue &&
                  gameInvitationModel?.IsConfirmed == false)  {
               await SendStringAsync(webSocket,   JsonConvert.

    SerializeObject(new
     { Result = "OK",
       Email = gameInvitationModel.InvitedBy,
        gameInvitationModel.EmailTo,
              gameInvitationModel.Id  }), ct);

       Task.Delay(500).Wait();
       gameInvitationModel = await gameInvitationService.Get(id);
      }
}
```


WOW! eBook
www.wowebook.org

12. Update the `Invoke` method in the communication middleware. This will work with email confirmations and game invitation confirmations from now on, with and without WebSockets:

```
public async Task Invoke(HttpContext context)
{
  if (context.WebSockets.IsWebSocketRequest)
  {
    ...
    switch (command.Operation.ToString())
    {
      ...
      case "CheckGameInvitationConfirmationStatus":
      { await
          ProcessGameInvitationConfirmation(context,webSocket, ct,
            command.Parameters.ToString());
                break; }
     }
  }
  else if (context.Request.Path.Equals
    ("/CheckEmailConfirmationStatus"))
  {  await ProcessEmailConfirmation(context);   }
  else if (context.Request.Path.Equals
    ("/CheckGameInvitationConfirmationStatus"))
  {  await ProcessGameInvitationConfirmation(context);  }
  else   {  await _next?.Invoke(context);  }
}
```

In this section, you have learned how to use method injection in your ASP.NET Core 3 web applications. This is the preferred method for injecting your services and you should use it whenever applicable.

You have advanced well with the implementation of the Tic-Tac-Toe game. Mostly everything around user registration, email confirmation, game invitation, and game invitation confirmation has now been implemented.

WOW! eBook
www.wowebook.org

Summary

In this chapter, you have learned about some more advanced concepts of ASP.NET Core 3 and implemented some of the missing components of the Tic-Tac-Toe application.

First, you created the client-side parts of the Tic-Tac-Toe web application using JavaScript. We have explored how to optimize our web applications by using bundling and minification, as well as WebSockets for real-time communication scenarios.

Furthermore, you have seen how to benefit from the integrated user and session handling, which was shown in an easy-to-understand example.

Then, we introduced globalization and localization for multilingual user interfaces, application and service configuration, as well as logging to better understand what is happening within our applications during runtime.

Finally, using a practical example, we illustrated how to build our applications once and then adapt them to different environments by using the concepts of multiple `ConfigureServices` and `Configure` methods, as well as multiple `Startup` classes, depending on deployment targets.

In the next chapter, we will introduce client-side development with Razor components, or Blazor, and we will deal with logging for our demo application.

WOW! eBook
www.wowebook.org

6
Introducing Razor Components and SignalR

We have so far seen a number of changes that have been introduced in ASP.NET Core 3 compared to previous versions of the framework, including the early-phase offerings of .NET Core. These have been mentioned in previous chapters through the use of our demo application, but it's now time to introduce what will be described by many as quite a significant introduction to ASP.NET Core 3: server-side Blazor, formerly known as **Razor components**.

In one section in `Chapter 5`, *Basic Concepts of ASP.NET Core 3: Part 2*, we explored at client-side development using JavaScript. For many developers who are used to the strong typing and other syntax benefits that come with working on Microsoft tech stacks, Blazor comes to the rescue. Blazor is an alternative to JavaScript.

Blazor integrates with .NET Core as server-side Blazor, and WebAssembly as client-side Blazor, and makes it possible to actually run C# directly on the browser and fully replace JavaScript.

WebAssembly (abbreviated to **Wasm**), as defined on its official page, `https://webassembly.org/`, is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for the compilation of high-level languages such as C/C++/Rust (and, in our case, C#), enabling deployment on the web for client and server applications.

Client-side Blazor works exclusively from the client-side. Currently, in preview, there is a prospect of it being shipped with later versions of ASP.NET Core.

Server-side Blazor at this point relies on existing technology, SignalR. A brief introduction is worthwhile, and we will duly cover what you need to know about it in a later section.

WOW! eBook
www.wowebook.org

You will learn how to build a simple Blazor application, and how it differs from normal ASP.NET Core 3. You will learn about the components that constitute a basic Blazor page. We finish the chapter by explaining logging and telemetry to help with debugging in production environments. You will learn about different options you have in logging important information for your application and you will learn how to configure your applications for logging using a file logger.

You will learn how to configure your application so you can run it in different hosting environments, be it in development, staging, or production.

The following topics will be covered in this chapter:

- Client-side development using C# Razor components
- Working with SignalR
- Using logging and telemetry for monitoring and supervision purposes
- Building once and running on multiple environments

Client-side development using C# Razor components

In its quest to provide a true full-stack experience on the .NET Core framework, Microsoft has been experimenting with client-side development using C# Razor components; at the time of writing, this is called server-side Blazor. There are future plans to release client-side Blazor, which runs directly on the WebAssembly, but that is beyond the scope of this book.

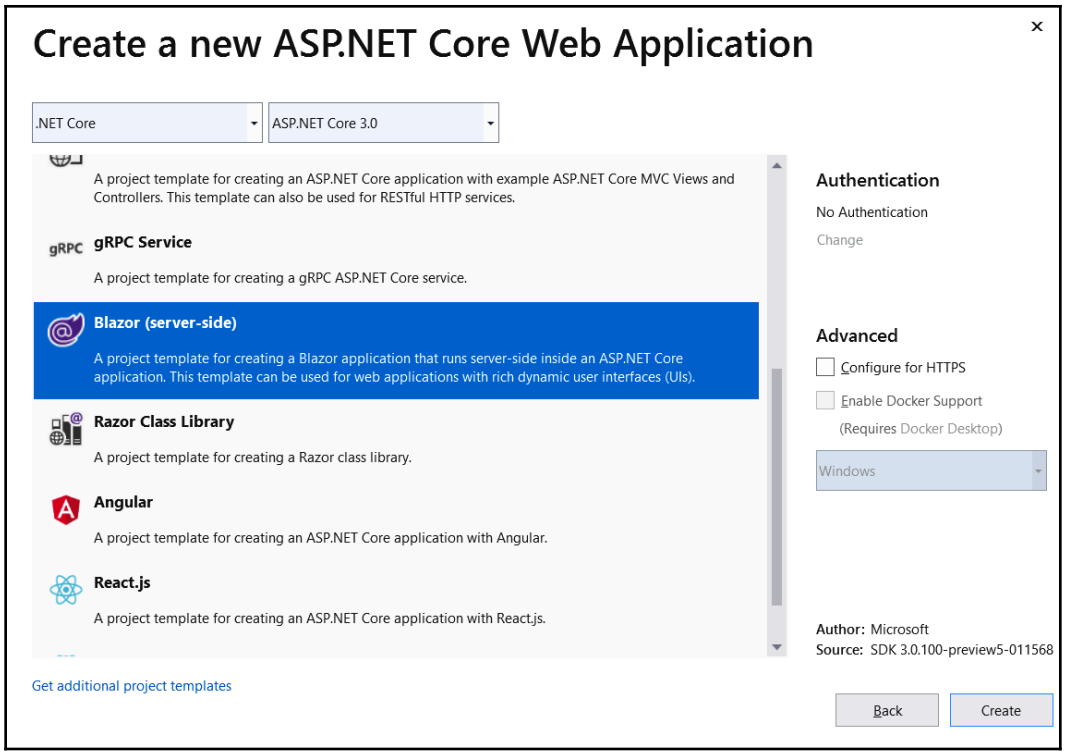
Microsoft initially released a C# Razor component template for ASP.NET Core 3, but this has now been renamed the Blazor (server-side) template.

Note that C# Razor and Blazor (server-side) components are essentially the same. Agreement was reached on using the name *server-side Blazor*, influenced by the long-running previously experimental Blazor project: `https://dotnet.microsoft.com/apps/aspnet/web-apps/client`

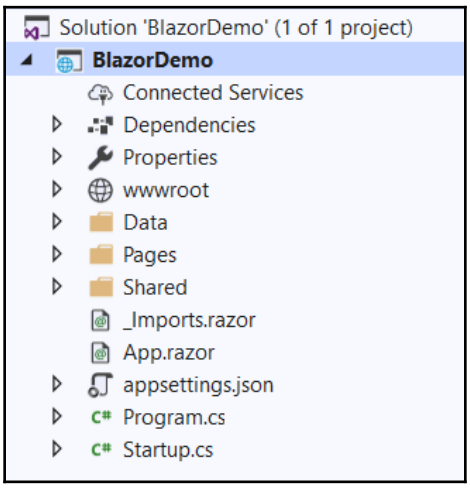
The name *server-side Blazor* can be misleading because Blazor is primarily meant to enhance client-side development with dynamic and rich user interfaces.

Let's take a break from our Tic-Tac-Toe demo application and create a simple web app using the server-side Blazor template. Create a new project using the ASP.NET Core application framework first and then the Blazor template as follows:

WOW! eBook
www.wowebook.org



When the web application is created, you will immediately notice that it has a project structure that is generally similar to any ASP.NET Core template with `Startup` and `Program` classes as follows:



WOW! eBook
www.wowebook.org

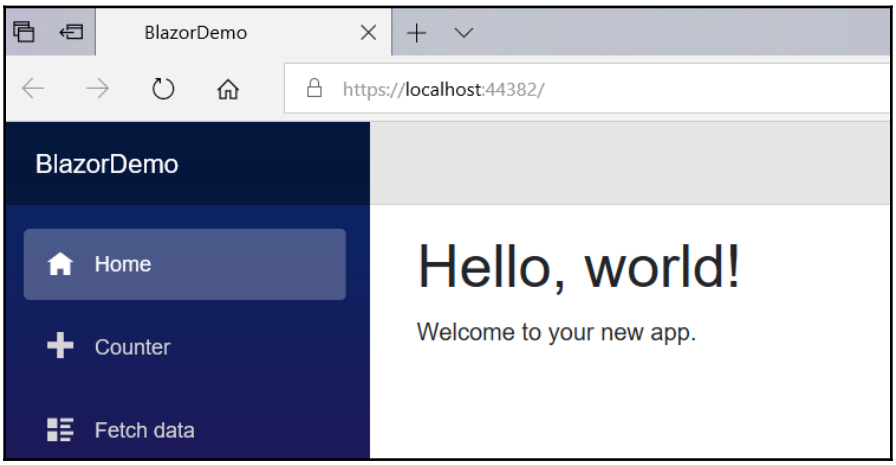
One difference that you will notice immediately is the new `.razor` file extension that is meant to identify any Razor components, for example, `App.razor`, as seen in the preceding screenshot. Another difference where we call the `AddServerSideBlazor()` method to the service collection is found in the `Startup` class in the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddServerSideBlazor();
    ...
}
```

In the previous code block, we can see that server-side Blazor is added as a service, and next, in the `Configure` method, endpoints for Blazor are configured, mainly in order to accept incoming connections for interactive components via the `MapBlazorHub()` method:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

Run the application as it is without changing anything, and you should be able to see the following in your browser:



The home page displays the **Hello, world!** text, which is defined in the `Index.razor` component, and likewise we have `Counter.razor` and `FetchData.razor` components that determine what is displayed when the respective tabs are clicked.

WOW! eBook
www.wowebook.org

As shared components for the whole app, we have `MainLayout.razor`, which is responsible for the application's layout, and `NavMenu.razor`, which defines navigation items on the left.

In this example template, the `Counter` component performs all its functionality by itself, but the `FetchData` component relies on an external C# service class to provide it with the data that it needs.

If you take a closer look at the counter functionality, most developers would expect to use JavaScript, but you will notice that there is no JavaScript in the `Counter` component. Take a closer look at any of the Razor components, and you will find no JavaScript or any reference to it! In fact, in the `wwwroot` section, for any traditional ASP.NET Core template, there will be a `js` folder, which is a glaring omission from our `BlazorDemo`! All this is by design; if you are like many backend C# developers who don't do too well with JavaScript, Blazor will be a relief to you!

Blazor is designed in such a way that you, as a developer, will be able to use C# **instead** of JavaScript on the client-side. It must be noted that Blazor can also work side by side with JavaScript.

Now, let's look at the following code block:

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" onclick="@IncrementCount">Click me</button>

@functions {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

The preceding code snippet represents the basic page component in a Blazor application. It all starts with routing, in which the exact URL for a page is specified after the `@page` directive. In the preceding example, you can access this page by adding `/counter` to the URL and will be able to get to this specific page.

WOW! eBook
www.wowebook.org

Before we give `body` some content, just under the `@page` directive, you can have `@using` directives, for example, `@using BlazorDemo.Data`, should you need to access another part of the application.

You can also have DI, where you can inject a service such as `@inject WeatherForecastService ForecastService` into a page should you need it.

Finally, we have the `@functions` section, where you can add as many C# functions as you may need, and this is normally where you would place your JavaScript functions.

Server-side Blazor is so named because, for all the preceding code to work, it is actually executed on the server. What communication channel does it use to ensure real-time rendering? We answer this question in the next section:

Working with SignalR

SignalR is a technology that powers the real-time functionality required for server-side Blazor. But first, let's try and understand what this technology really is, and how everything fits together.

What is SignalR

SignalR existed prior to the introduction of the ASP.NET Core family of framework versions, and it was simply designed as a library to cater to real-time communication between the server and its clients.

It leveraged and improved upon the use of the WebSockets technology, which we covered in the previous chapter. If you read around the **World Wide Web** (**WWW**), you will notice that most examples for SignalR explain its usage on real-time chat applications, and rightly so, but there are a lot of applicable situations where it can be utilized as well, including dashboards, real-time stock trade applications, and so on.

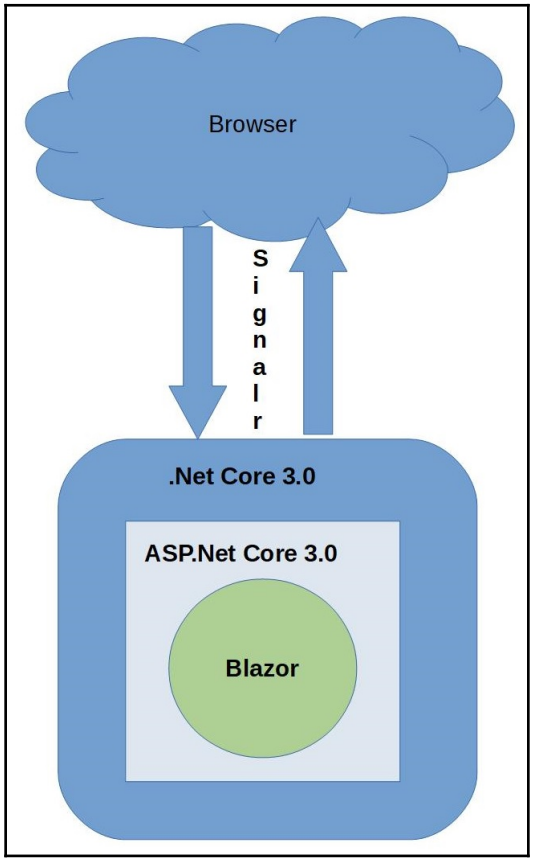
SignalR has been evolving and maturing as a technology, and consequently it works together with Blazor or Razor components, whose client-side nature relies on robust, but powerful, communication with the server. We explain how this technology powers server-side Blazor in the next section.

WOW! eBook
www.wowebook.org

SignalR with server-side Blazor or Razor components

We have had a brief look at server-side Blazor, but one thing we need to always have in mind is the fact that server-side Blazor uses SignalR to push content from the server to the client-side instantaneously. SignalR is a library that was designed to deal with situations where we need real-time, client-side, and server interactions in web applications, using **hubs**.

There is a well-documented usage of SignalR for chat applications, but a simple guide as to when to use SignalR is to look at scenarios where there is a need for a lot of updates from the server. In the case of server-side Blazor, there are a lot of updates between the client and the server, all channeled through SignalR. The following diagram illustrates this:



WOW! eBook
www.wowebook.org

For this book, which concerns ASP.NET Core 3, it's enough just to know the ecosystem surrounding the use of SignalR. Most SignalR implementations you will come across actually work in the background (as in the case of server-side Blazor), abstracted from you, and therefore we will not delve into how to use it.

Using logging and telemetry for monitoring and supervision purposes

When you are developing your applications, you use one of the well-known integrated development environments such as Visual Studio 2019 or Visual Studio Code, as described in the initial chapters of the book. You do this every day, and most of the things you do become second nature and you perform them automatically after some time.

It is natural for you to be able to debug your applications and understand what is happening during runtime by using the advanced debugging features of Visual Studio 2019, for example. Looking up variable values, seeing what methods get called in what order, understanding what instances are injected, and capturing exceptions, are key to building applications that are robust and respond to business needs.

Then, when deploying your applications to production environments, you suddenly miss all of those features. Rarely will you find a production environment where Visual Studio is installed, but errors and unexpected behaviors will happen and you will need to be able to understand and fix them as fast as possible.

That is where logging and telemetry come into their own. By instrumenting your applications and logging when entering and leaving methods, as well as important variable values or any kind of information you consider important during runtime, you will be able to go to the application log and see what is happening in the production environment in the event of issues.

In this section, we go back to our Tic-Tac-Toe demo application, where we are going to show you how to use logging and exception handling to provide an industrialized solution to the problem where we only get an exception in production, without the finer details that can help you to debug the issue.

WOW! eBook
www.wowebook.org

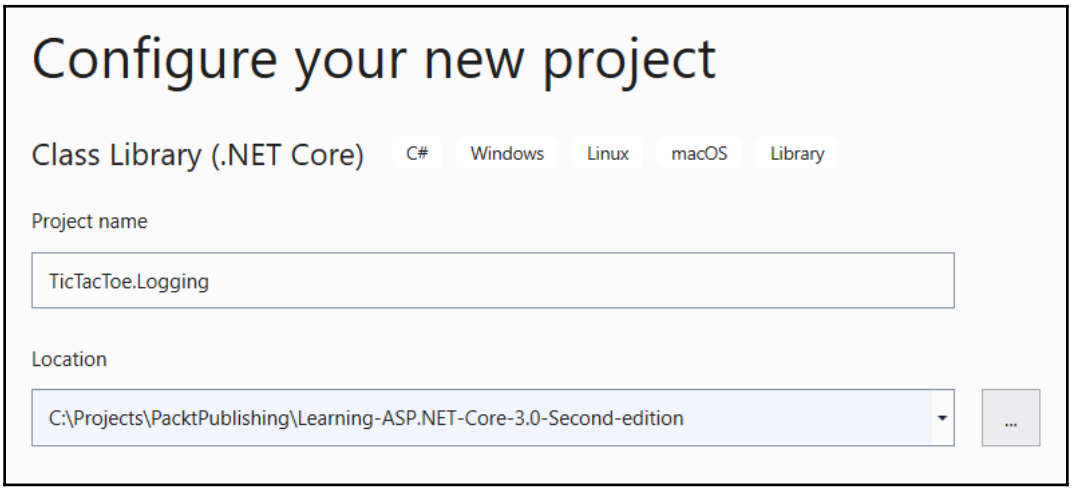
ASP.NET Core 3 provides built-in support for logging to the following targets:

- Azure App Services
- Console
- Windows event source
- Trace
- Debugger output
- Application Insights

However, files, databases, and logging services are not supported by default. If you want to send your logs to these targets, you need to use a third-party logger solution such as Log4net, Serilog, NLog, Apache, ELMAH, or Loggr.

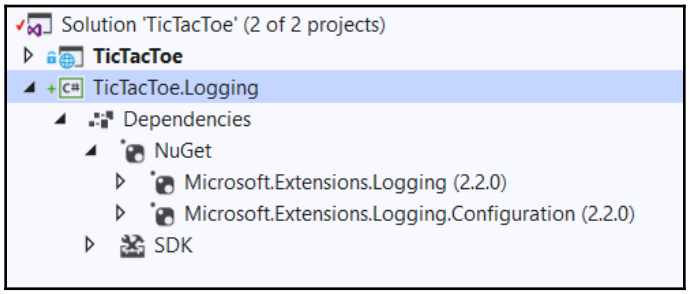
You can also easily create your own provider by implementing the `ILoggerProvider` interface, as follows:

1. Add a new **Class Library (.NET Core)** project to the solution and call it `TicTacToe.Logging` (delete the autogenerated `Class1.cs` file):

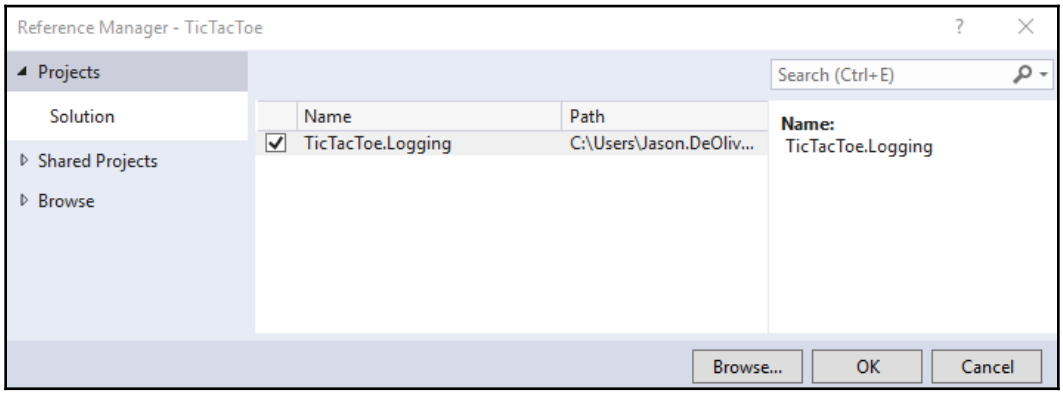


WOW! eBook
www.wowebook.org

2. Add the `Microsoft.Extensions.Logging` and `Microsoft.Extensions.Logging.Configuration` NuGet packages, via NuGet Package Manager:



3. Add a project reference from the `TicTacToe` web application project so that we can use assets from the `TicTacToe.Logging` class library:



4. Add a new **POCO** (short for **Plain Old CLR Object**) class called `LogEntry` to the **TicTacToe.Logging** project. This will contain log data, with an event `id`, the message for the actual log, the log level, the level (information, warning, or critical), and finally a timestamp when a log is created:

```
public class LogEntry
{
  public int EventId { get; internal set; }
  public string Message { get; internal set; }
  public string LogLevel { get; internal set; }
  public DateTime CreatedTime { get; internal set; }
}
```


WOW! eBook
www.wowebook.org

5. Add a new class called `FileLoggerHelper`, which will be used for file operations. Then we add field definitions and a constructor. The constructor makes sure that, every time `FileLoggerHelper` is instantiated, it is forced to accept a filename and make it available for use in internal methods such as `InsertLog`, as follows:

```
public  class FileLoggerHelper
{
  private string fileName;

  public FileLoggerHelper(string fileName)
  {
    this.fileName = fileName;
  }

  static ReaderWriterLock locker = new ReaderWriterLock();

  //....
  }

}
```

Let's then add an `InsertLog` method to the `FileLoggerHelper` class. The method creates a file directory if it doesn't already exist, logs events to a file after acquiring a lock, and then releases them after use. `InsertLog` is implemented as follows:

```
public void InsertLog(LogEntry logEntry)
{
  var directory = System.IO.Path.GetDirectoryName(fileName);

  if (!System.IO.Directory.Exists(directory))
     System.IO.Directory.CreateDirectory(directory);

  try
  {
    locker.AcquireWriterLock(int.MaxValue);
    System.IO.File.AppendAllText(fileName,
    $"{logEntry.CreatedTime} {logEntry.EventId} {logEntry.LogLevel}
  {
    logEntry.Message}" + Environment.NewLine);
  }
  finally
  {
    locker.ReleaseWriterLock();
  }
```


WOW! eBook
www.wowebook.org

Add a new class called `FileLogger` and implement the `ILogger` interface. The file logger concrete class will allow us to make use of the logging functionality that is available in the `ILogger` interface template provided by Microsoft in the .NET Core framework:

```
public sealed class FileLogger : ILogger
{
 public IDisposable BeginScope<TState>(TState state)
 {
   return null;
 }
 public bool IsEnabled(LogLevel logLevel)
 {
   return (_filter == null || _filter(_categoryName, logLevel));
 }
 public void Log<TState>(LogLevel logLevel, EventId eventId,
 TState state, Exception exception,             Func<TState,
Exception, string> formatter)
 {
     throw new NotImplementedException();
 }
}
```

Before we implement the `Log` method, let's create our constructor and field definitions. We make sure that the category name, log level, and filename are all supplied, and we also create a new instance of `FileLoggerHelper` as follows:

```
private string _categoryName;
private Func<string, LogLevel, bool> _filter;
private string _fileName;
private FileLoggerHelper _helper;

public FileLogger(string categoryName, Func<string,
  LogLevel,
 bool> filter, string fileName)
{
  _categoryName = categoryName;
  _filter = filter;
  _fileName = fileName;
  _helper = new FileLoggerHelper(fileName);
}
```


WOW! eBook
www.wowebook.org

And then there is our main `Log` method, now implemented as follows:

```
public void Log<TState>(LogLevel logLevel, EventId eventId, TState
state, Exception exception,     Func<TState, Exception, string>
formatter)
{
   if (!IsEnabled(logLevel)) return;
   if (formatter == null) throw new
     ArgumentNullException(nameof(formatter));
   var message = formatter(state, exception);
   if (string.IsNullOrEmpty(message)) return;
   if (exception != null) message += "\n" + exception.ToString();
   var logEntry = new LogEntry
   {
      Message = message,
      EventId = eventId.Id,
      LogLevel = logLevel.ToString(),
      CreatedTime = DateTime.UtcNow
   };
   _helper.InsertLog(logEntry);
}
```

6. Add a new class called `FileLoggerProvider` and implement the `ILoggerProvider` interface. This is used to supply `FileLogger` instances of `ILogger` when required by ASP.NET Core, and will be injected later:

```
          public class FileLoggerProvider : ILoggerProvider
          {
            private readonly Func<string, LogLevel, bool> _filter;
            private string _fileName;

            public FileLoggerProvider(Func<string, LogLevel, bool>
              filter, string fileName)
            {
              _filter = filter;
              _fileName = fileName;
            }

            public ILogger CreateLogger(string categoryName)
            {
              return new FileLogger(categoryName, _filter,
_fileName);
            }

            public void Dispose()   { }
          }
```


WOW! eBook
www.wowebook.org

7. To simplify calling the **file logging provider** from the web application, we need to add a static class called `FileLoggerExtensions` (with the configuration section, filename, and log verbosity level as parameters):

```
public static class FileLoggerExtensions
{
  const long DefaultFileSizeLimitBytes = 1024 * 1024 *
   1024;
  const int DefaultRetainedFileCountLimit = 31;
}
```

Our `FileLoggerExtensions` class will have three different overloads on the `AddFile` method. Now, let's add our first implementation of the `AddFile` method:

```
public static ILoggingBuilder AddFile(this ILoggingBuilder
loggerBuilder, IConfigurationSection configuration)
{
  if (loggerBuilder == null) throw new
    ArgumentNullException(nameof(loggerBuilder))
  if (configuration == null) throw new
    ArgumentNullException(nameof(configuration));
  var minimumLevel = LogLevel.Information;
  var levelSection = configuration["Logging:LogLevel"];
  if (!string.IsNullOrWhiteSpace(levelSection))
  {
   if (!Enum.TryParse(levelSection, out minimumLevel))
   {
   System.Diagnostics.Debug.WriteLine("The minimum level setting
     `{0}` is invalid", levelSection);
   minimumLevel = LogLevel.Information;
   }
  }
  return loggerBuilder.AddFile(configuration[
      "Logging:FilePath"], (category, logLevel) =>  (logLevel >=
        minimumLevel), minimumLevel);
 }
```

And then there is the second overload for the `AddFile` method:

```
public static ILoggingBuilder AddFile(this ILoggingBuilder
         loggerBuilder, string filePath, Func<string, LogLevel,
         bool> filter, LogLevel minimumLevel =
          LogLevel.Information)
        {
            if (String.IsNullOrEmpty(filePath)) throw
             new ArgumentNullException(nameof(filePath));
```


WOW! eBook
www.wowebook.org

```
            var fileInfo = new System.IO.FileInfo(filePath);

            if (!fileInfo.Directory.Exists)
                fileInfo.Directory.Create();

            loggerBuilder.AddProvider(new FileLoggerProvider
             (filter, filePath));

            return loggerBuilder;
        }
```

Then, there is the third overload implementation for the `AddFile` method:

```
public static ILoggingBuilder AddFile(this ILoggingBuilder
         loggerBuilder, string filePath,  LogLevel minimumLevel =
          LogLevel.Information)
        {
            if (String.IsNullOrEmpty(filePath)) throw
             new ArgumentNullException(nameof(filePath));

            var fileInfo = new System.IO.FileInfo(filePath);

            if (!fileInfo.Directory.Exists)
                fileInfo.Directory.Create();

            loggerBuilder.AddProvider(new FileLoggerProvider
            ((category,
             logLevel) => (logLevel >= minimumLevel), filePath));

            return loggerBuilder;
        }
```

8. In the `TicTacToe` web project, add two new options, called `LoggingProviderOption` and `LoggingOptions`, to the `Options` folder:

```
public class LoggingProviderOption
{
  public string Name { get; set; }
  public string Parameters { get; set; }
  public int LogLevel { get; set; }
}
public class LoggingOptions
{
  public LoggingProviderOption[] Providers { get; set; }
}
```


WOW! eBook
www.wowebook.org

9. In the `TicTacToe` web project, add a new extension called `ConfigureLoggingExtension` to the `Extensions` folder:

```
public static class ConfigureLoggingExtension
{
  public static ILoggingBuilder
AddLoggingConfiguration(this
   ILoggingBuilder loggingBuilder, IConfiguration
    configuration)
  {
    var loggingOptions = new LoggingOptions();
    configuration.GetSection("Logging").
     Bind(loggingOptions);

    foreach (var provider in loggingOptions.Providers)
    {
      switch (provider.Name.ToLower())
      {
        case "console": {  loggingBuilder.AddConsole();
        break;
      }
        case "file": {
          string filePath = System.IO.Path.Combine(
            System.IO.Directory.GetCurrentDirectory(),
             "logs",
             $"TicTacToe_{System.DateTime.Now.ToString(
                "ddMMyyHHmm")}.log");
          loggingBuilder.AddFile(filePath,
            (LogLevel)provider.LogLevel);
          break;
        }
        default:  {  break;   }
      }
    }
    return loggingBuilder;
  }
}
```

10. Go to the `Program` class of the `TicTacToe` web application project, update the `BuildWebHost` method, and call the extension:

```
public static IHostBuilder CreateHostBuilder(string[] args)
  =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.CaptureStartupErrors(true);
```


WOW! eBook
www.wowebook.org

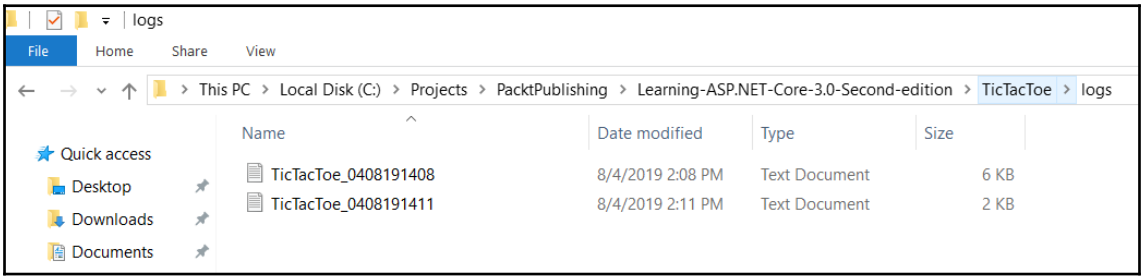
```
                webBuilder.PreferHostingUrls(true);
                webBuilder.UseUrls("http://localhost:5000");
                webBuilder.ConfigureLogging((hostingcontext,
                 logging) =>
                {
                    logging.AddLoggingConfiguration(
                     hostingcontext.Configuration);
                });
            });
```

Don't forget to add the following `using` statement at the beginning of the class:
`using TicTacToe.Extensions;`.

11. Add a new section called `Logging` to the `appsettings.json` file:

```
"Logging": {
  "Providers": [
    {
      "Name": "Console",
      "LogLevel": "1"
    },
    {
      "Name": "File",
      "LogLevel": "2"
    }
  ],
  "MinimumLevel": 1
}
```

12. Start the application and verify that a new log file has been created in a folder called `logs` within the application folder:



WOW! eBook
www.wowebook.org

This is the first step and is straightforward and quick to complete. You now have a log file to which you can write your logs. You will see that it is just as easy to use the integrated logging functionalities to create logs from anywhere within your ASP.NET Core 3 applications (`Controllers`, `Services`, and more).

Let's quickly add some logs to the Tic-Tac-Toe application:

1. Update the `UserRegistrationController` constructor implementation so that we supply a logger instance for the entire controller:

```
readonly IUserService _userService;
readonly IEmailService _emailService;
readonly ILogger<UserRegistrationController> _logger;
public UserRegistrationController(IUserService userService,
 IEmailService emailService,
  ILogger<UserRegistrationController>
 logger)
{
  _userService = userService;
  _emailService = emailService;
  _logger = logger;
}
```

2. Update the `EmailConfirmation` method in `UserRegistrationController` and add a log at the start of the method:

```
_logger.LogInformation($"##Start## Email confirmation
    process for {email}");
```

3. Update the `EmailService` implementation, and add a logger to its constructor so that it is available to the email service:

```
public class EmailService : IEmailService
        {
          private EmailServiceOptions _emailServiceOptions;
          readonly ILogger<EmailService> _logger;
          public EmailService(IOptions<EmailServiceOptions>
           emailServiceOptions, ILogger<EmailService> logger)
          {
            _emailServiceOptions = emailServiceOptions.Value;
            _logger = logger;
          }
 }
```


WOW! eBook
www.wowebook.org

And then replace the `SendMail` method in `EmailService` with the following:

```
public Task SendEmail(string emailTo, string subject,
  string message)
{
  try
  {
    _logger.LogInformation($"##Start sendEmail## Start
      sending Email to {emailTo}");

    using (var client =  new
      SmtpClient(_emailServiceOptions.MailServer,
      int.Parse(_emailServiceOptions.MailPort)))
    {
      if (bool.Parse(_emailServiceOptions.UseSSL)
       == true)
       client.EnableSsl = true;

      if (!string.IsNullOrEmpty
      (_emailServiceOptions.UserId))
          client.Credentials =  new NetworkCredential
           (_emailServiceOptions.UserId,
           _emailServiceOptions.Password);

        client.Send(new MailMessage
      ("ken@afrikancoder.co.za", emailTo, subject,
       message));
    }
  }
  catch (Exception ex) { _logger.LogError($"Cannot
   send email {ex}");  }

  return Task.CompletedTask;
}
```


WOW! eBook
www.wowebook.org

4. Then, open the generated log file and analyze its contents, after running the application and registering a new user:

```
8/4/2019 12:37:28 PM 1 Information  Executing RedirectResult, redirecting to /UserRegistration/EmailConfirmation?Email=ken@afrikancoder.co.za.
8/4/2019 12:37:28 PM 2 Information  Executed action TicTacToe.Controllers.UserRegistrationController.Index (TicTacToe) in 166.9566ms
8/4/2019 12:37:28 PM 1 Information  Executed endpoint 'TicTacToe.Controllers.UserRegistrationController.Index (TicTacToe)'
8/4/2019 12:37:28 PM 2 Information  Request finished in 307.069ms 302
8/4/2019 12:37:28 PM 1 Information  Request starting HTTP/1.1 GET http://localhost:64667/UserRegistration/EmailConfirmation?Email=ken@afrikancoder.co.za
8/4/2019 12:37:28 PM 1 Warning  CultureProviderResolverService returned the following unsupported cultures ''.
8/4/2019 12:37:28 PM 2 Warning  CultureProviderResolverService returned the following unsupported UI Cultures ''.
8/4/2019 12:37:28 PM 0 Information  Executing endpoint 'TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe)'
8/4/2019 12:37:28 PM 3 Information  Route matched with {action = "EmailConfirmation", controller = "UserRegistration"}. Executing controller action with signature System.
8/4/2019 12:37:28 PM 1 Information  Executing action method TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe) - Validation state: Valid
8/4/2019 12:37:28 PM 0 Information  ##Start## Email confirmation  process for ken@afrikancoder.co.za
8/4/2019 12:37:28 PM 0 Information  ##Start sendEmail## Start sending Email to ken@afrikancoder.co.za
8/4/2019 12:37:31 PM 0 Error  Cannot send email System.Net.Mail.SmtpException: Failure sending mail.
 ---> System.Net.Internals.SocketExceptionFactory+ExtendedSocketException (10061): No connection could be made because the target machine actively refused it. [::ffff:127.
   at System.Net.Sockets.Socket.DoConnect(EndPoint endPointSnapshot, SocketAddress socketAddress)
   at System.Net.Sockets.Socket.Connect(EndPoint remoteEP)
   at System.Net.Sockets.TcpClient.Connect(IPEndPoint remoteEP)
   at System.Net.Sockets.TcpClient.Connect(String hostname, Int32 port)
--- End of stack trace from previous location where exception was thrown ---
   at System.Net.Sockets.TcpClient.Connect(String hostname, Int32 port)
   at System.Net.Mail.SmtpConnection.GetConnection(String host, Int32 port)
   at System.Net.Mail.SmtpTransport.GetConnection(String host, Int32 port)
   at System.Net.Mail.SmtpClient.GetConnection()
   at System.Net.Mail.SmtpClient.Send(MailMessage message)
   --- End of inner exception stack trace ---
   at System.Net.Mail.SmtpClient.Send(MailMessage message)
   at TicTacToe.Services.EmailService.SendEmail(String emailTo, String subject, String message) in C:\Projects\PacktPublishing\Learning-ASP.NET-Core-3.0-Second-edition\Tic
8/4/2019 12:37:31 PM 2 Information  Executed action method TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe), returned result Microsoft.AspNet
8/4/2019 12:37:31 PM 1 Information  Executing ViewResult, running view EmailConfirmation.
8/4/2019 12:37:31 PM 4 Information  Executed ViewResult - view EmailConfirmation executed in 32.5416ms.
8/4/2019 12:37:31 PM 2 Information  Executed action TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe) in 2857.3986ms
8/4/2019 12:37:31 PM 1 Information  Executed endpoint 'TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe)'
8/4/2019 12:37:31 PM 2 Information  Request finished in 3042.8595ms 200 text/html; charset=utf-8
8/4/2019 12:37:38 PM 1 Information  Request starting HTTP/1.1 GET http://localhost:64667/CheckEmailConfirmationStatus
8/4/2019 12:37:39 PM 1 Information  Request starting HTTP/1.1 GET http://localhost:64667/GameInvitation?email=ken@afrikancoder.co.za
8/4/2019 12:37:39 PM 1 Warning  CultureProviderResolverService returned the following unsupported cultures ''.
8/4/2019 12:37:39 PM 2 Warning  CultureProviderResolverService returned the following unsupported UI Cultures ''.
8/4/2019 12:37:39 PM 0 Information  Executing endpoint '405 HTTP Method Not Supported'
8/4/2019 12:37:39 PM 1 Information  Executed endpoint '405 HTTP Method Not Supported'
8/4/2019 12:37:39 PM 2 Information  Request finished in 101.9243ms 405
```

You will notice that the start of the email confirmation process and the start of sending an email have all been duly recorded in the logs. The failure to send an email itself has also been logged as an exception with its stack trace.

Building once and running on multiple environments

After building your applications, you have to think about deploying them to different environments. As you have already seen in the previous section on configuration, you can use configuration files to change the configuration of your services and even your application.

In the case of multiple environments, you have to duplicate the `appsettings.json` file for each environment and name it accordingly: `appsettings.{EnvironmentName}.json`.

WOW! eBook
www.wowebook.org

ASP.NET Core 3 will automatically retrieve configuration settings in hierarchical order, first from the common `appsettings.json` file and then from the corresponding `appsettings.{EnvironmentName}.json` file, while adding or replacing values if necessary.

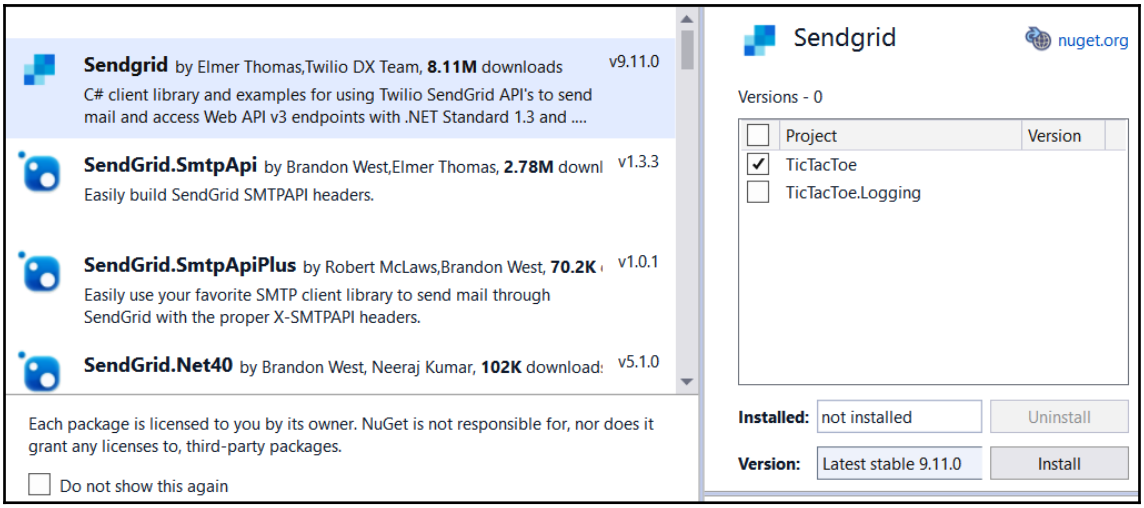
However, developing conditional code that uses different components based on different deployment environments and configurations seems to be complicated at first. In traditional applications, you must create a lot of code to handle all of the different operations by yourself and then maintain it.

In ASP.NET Core 3, you have a vast number of internal functionalities at your disposal to achieve this goal. You can then simply use environment variables (development, staging, production, and more) to indicate a specific runtime environment, thus configuring your application for that environment.

As you will see during this section, you can use specific method names and even class names to use existing injection and override mechanisms, provided by ASP.NET Core 3 out of the box, to configure your applications.

In the following example, we are adding an environment-specific component (**SendGrid**) to the application, which only has to be used if the application is deployed to a specific production environment (Azure):

1. Add the **Sendgrid** NuGet package to the project. This will be used for future Azure production deployments of the Tic-Tac-Toe application:



WOW! eBook
www.wowebook.org

2. Add a new service called `SendGridEmailService` within the `Services` folder. This will be used to send emails via `SendGrid`. Have it inherit the `IEmailService` interface and implement the specific `SendEmail` method. Firstly, the constructor:

```
public class SendGridEmailService : IEmailService
{
  private EmailServiceOptions _emailServiceOptions;
  private ILogger<EmailService> _logger;
  public SendGridEmailService(IOptions<EmailServiceOptions>
   emailServiceOptions, ILogger<EmailService> logger)
  {
    _emailServiceOptions = emailServiceOptions.Value;
    _logger = logger;
  }
 //....
}
```

3. And then add a `SendMail` method to the same `SendGridEmailService` class:

```
public Task SendEmail(string emailTo, string subject, string
  message)
{
  _logger.LogInformation($"##Start## Sending email via
             SendGrid to :{emailTo} subject:{subject} message:
              {message}");
  var client = new SendGrid.SendGridClient(_emailServiceOptions.
    RemoteServerAPI);
  var sendGridMessage = new SendGrid.Helpers.Mail.SendGridMessage
  {
    From = new SendGrid.Helpers.Mail.EmailAddress(
     _emailServiceOptions.UserId)
  };
  sendGridMessage.AddTo(emailTo);
  sendGridMessage.Subject = subject;
  sendGridMessage.HtmlContent = message;
  client.SendEmailAsync(sendGridMessage);
  return Task.CompletedTask;
}
```

4. Add a new extension method to more easily declare specific email services for specific environments. For that, go to the `Extensions` folder and add a new `EmailServiceExtension` class:

```
public static class EmailServiceExtension
{
  public static IServiceCollection AddEmailService(
```


WOW! eBook
www.wowebook.org

```
        this IServiceCollection services, IHostingEnvironment
         hostingEnvironment, IConfiguration configuration)
      {
        services.Configure<EmailServiceOptions>
         (configuration.GetSection("Email"));
        if (hostingEnvironment.IsDevelopment() ||
            hostingEnvironment.IsStaging())
        {
          services.AddSingleton<IEmailService, EmailService>();
        }
        else
        {
          services.AddSingleton<IEmailService,
            SendGridEmailService>();
        }
        return services;
      }
    }
```

5. Update the `Startup` class to use the created assets. For better readability and maintainability, we will go even further and create a dedicated `ConfigureServices` method for each environment we have to support, remove the existing `ConfigureServices` method, and add the following environment-specific `ConfigureServices` methods. First, we configure the definitions and constructor:

```
public IConfiguration _configuration { get; }
        public IHostingEnvironment _hostingEnvironment { get; }
        public Startup(IConfiguration configuration,
         IHostingEnvironment hostingEnvironment)
        {
          _configuration = configuration;
          _hostingEnvironment = hostingEnvironment;
        }
```

Secondly, we configure common services:

```
public void ConfigureCommonServices(IServiceCollection services)
{
  services.AddLocalization(options => options.ResourcesPath =
   "Localization");
  services.AddMvc().AddViewLocalization(
    LanguageViewLocationExpanderFormat.Suffix, options =>
            options.ResourcesPath =
"Localization").AddDataAnnotationsLocalization();
  services.AddSingleton<IUserService, UserService>();
  services.AddSingleton<IGameInvitationService,
```


WOW! eBook
www.wowebook.org

```
    GameInvitationService>();
    services.Configure<EmailServiceOptions>
     (_configuration.GetSection("Email"));
    services.AddEmailService(_hostingEnvironment, _configuration);
    services.AddRouting();
    services.AddSession(o =>
    {
       o.IdleTimeout = TimeSpan.FromMinutes(30);
    });
   }
```

And finally, we configure a few specific services:

```
public void ConfigureDevelopmentServices(
 IServiceCollection services)
{
  ConfigureCommonServices(services);
}

public void ConfigureStagingServices(
 IServiceCollection services)
{
  ConfigureCommonServices(services);
}

public void ConfigureProductionServices(
 IServiceCollection services)
{
  ConfigureCommonServices(services);
}
```

Note that you could also apply the same approach to the `Configure` method in the `Startup` class. For that, you just remove the existing `Configure` method and add new methods for the environments you would like to support, such as `ConfigureDevelopment`, `ConfigureStaging`, and `ConfigureProduction`. The best practice would be to combine all common code into a `ConfigureCommon` method and call it from the other methods, as shown here for specific `ConfigureServices` methods.

6. Start the application by pressing *F5* and verify that everything is still running correctly. You should see that the added methods will automatically be used and that the application is fully functional.

WOW! eBook
www.wowebook.org

That was easy and straightforward! No specific conditional code for the environments, nothing complicated to evolve and to maintain; just very clear and easy-to-understand methods that contain the environment name they have been developed for. This constitutes a very clean solution to the problem of building once and running on multiple environments.

But that is not all! What if we told you that you do not need to have a single `Startup` class? What if you could have a dedicated `Startup` class for each environment with only the code applicable to its context? That would be great, right? Well, that is exactly what ASP.NET Core 3 provides.

To be able to use dedicated `Startup` classes for each environment, you just have to update the `Program` class, the main entry point for ASP.NET Core 3 applications. You change a single line in the `BuildWebHost` method to pass the assembly name `.UseStartup("TicTacToe")` instead of `.UseStartup<Startup>()`, and then you can use this fantastic feature:

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
    .UseStartup("TicTacToe")
    .PreferHostingUrls(true)
    .UseUrls("http://localhost:5000")
    .UseApplicationInsights()
    .Build();
  }
}
```

Now, you can add dedicated `Startup` classes for different environments, such as `StartupDevelopment`, `StartupStaging`, and `StartupProduction`. As with the method approach before, they will be used automatically; nothing else needs to be done on your side. Just update the `Program` class, implement your environment-specific `Startup` classes, and it works. ASP.NET Core 3 really makes our lives much easier by providing these useful features.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we were introduced to server-side Blazor, which, in all aspects, deserves a book on its own, and you are encouraged to find extra reading material on Blazor should you want to be proficient in using this new addition to ASP.NET Core. We looked at how to create a basic application with server-side Blazor, and we looked at the most important components for every Blazor application.

We then introduced SignalR as one of the underlying technologies that make server-side Blazor work, and again, additional reading is recommended for more advanced readers. For most developers, just having an awareness of how SignalR fits within the Blazor ecosystem will be fine for now.

We had a look at telemetry and logging to help us with troubleshooting problems when we publish our applications in production. We learned in detail how to configure and use file logging.

Lastly, we introduced you to the fact that you can change settings, and even configure what services you want to run, depending on the environment.

Having had a walkthrough and a high-level view of most important building blocks for ASP.NET Core 3, all the way from the first chapter, we are now well placed to go into somewhat more detail, starting with ASP.NET Core MVC in the next chapter.

WOW! eBook
www.wowebook.org

7
Creating ASP.NET Core MVC Applications

Most of today's modern web applications are based on the **Model View Controller** pattern, also commonly called **MVC**. You may have noticed that we also used it in the previous chapters to build the foundations of the Tic-Tac-Toe demo application. So, you have already worked with the MVC architecture in multiple places, without even knowing what was happening in the background and why it was important to apply this specific pattern.

Since its first release in 2007, the ASP.NET MVC framework has proven itself over the years, until effectively becoming the market standard. Microsoft has successfully evolved it into an industrialized and efficient framework with high developer productivity. There are many examples of web applications that take full advantage of the multiple features MVC has to offer. One great example is Stack Overflow. It provides information to developers and has a very high user base, with the need to scale to thousands, or even millions, of users at the same time.

In this chapter, you will acquire the skills that will allow you to create an MVC application and ascertain what kind of device is accessing your application. You will also learn how to use Tag Helpers and create View Components and partial views, and will also be equipped with the skills you need to be able to create unit and integration tests.

First, we will start by dissecting what an MVC application consists of from a high-level view and then go a bit deeper into the finer details, such as view pages and components. Then, we will look at view engines, how to structure projects, and layering our project.

WOW! eBook
www.wowebook.org

In this chapter, we will cover the following topics:

- Understanding the Model View Controller pattern
- Creating dedicated layouts for multiple devices
- Understanding ASP.NET Core state management
- Using view pages, partial views, View Components, and Tag Helpers
- Dividing a web application into multiple areas
- Applying advanced concepts such as view engines, unit tests, and integration tests
- Layering ASP.NET Core 3 applications

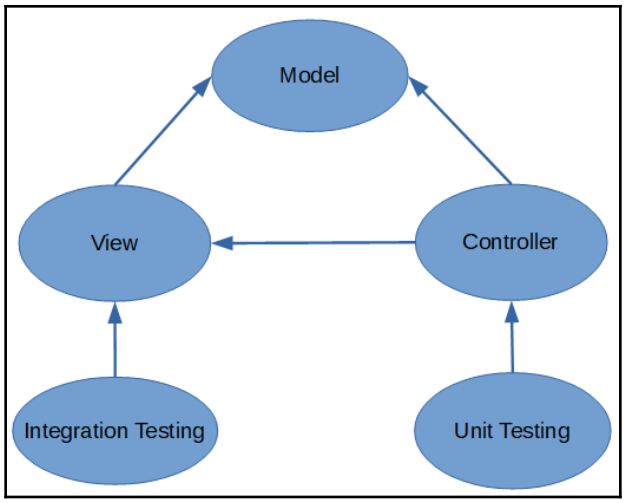
Understanding the Model View Controller pattern

The **MVC** pattern separates applications into three main layers: models, views, and controllers. One of the benefits of this pattern is the **separation of concerns** (**SoC**), which can also be described as the **Single Responsibility Principle** (**SRP**), which makes it possible for us to develop, debug, and test application features independently.

In software engineering, it is considered good practice to keep similar functionality as a unit. Mixing different responsibilities in a unit is considered an anti-pattern. There are other terms, such as **heavy coupling**, that describe a similar scenario where, for example, changing one aspect of a class requires changes in others to create a ripple effect. To avoid this effect, concepts of SoC and SRP were coined and are encouraged as they greatly help in unit testing as well, making sure the required functionality does what it purports to do.

When using the **MVC** pattern, a user request is routed to a **Controller**, which will use a **Model** to retrieve data and perform actions. The **Controller** selects a corresponding **View** for the user while providing it with the necessary data from the **Model**. There is less of an impact if a layer (for example, the **Views** layer) changes since it is now loosely coupled to the other layers of your applications (for example, controllers and models). It is also much easier to test the different layers of your applications. In the end, you will have better maintainability and more robust code by using this pattern:

WOW! eBook
www.wowebook.org



It must be mentioned that this is a simplistic diagram and shows the main concentration areas for unit testing and integration testing. A better understanding of each component and how it relates to the others will be provided in the following sections.

Models

A model contains the logical data structures as well as the data of your applications, independent of their visual representations. In the context of ASP.NET Core 3, it also supports localization and validation, as you have seen in the previous chapters.

Models can be created in the same project with your views and controllers or in a dedicated project for better organization of our solution project.

In ASP.NET Core 3, scaffolding uses models to auto-generate views. Furthermore, models can be used to bind forms to entity objects automatically. In terms of data persistence, various data storage targets can be used. In the case of databases, you should be using Entity Framework Core's **object-relational mapper** (**ORM**), which will be introduced in `Chapter 9`, *Accessing Data Using Entity Framework Core 3*. Models are serialized when we work with web APIs.

WOW! eBook
www.wowebook.org

Views

A view provides a visual representation and user interface elements of your applications. When using ASP.NET Core 3, views are written using HTML, Razor markup, and Razor components. Views generally have a `.cshtml` file extension and in the case of using the *Blazor template*, which we introduced in `Chapter 6`, *Introducing Razor Components and SignalR*, they have a `.razor` file extension.

A view either contains a complete web page, a web page part (called a partial view), or a layout. In ASP.NET Core 3, a view can be separated into logical subdivisions with their own behaviors, which are called **View Components**.

Additionally, **Tag Helpers** allow you to centralize and encapsulate HTML code in a single tag and use it across all your applications.

ASP.NET Core 3 already includes many already existing Tag Helpers that improve developer productivity.

Controllers

A controller manages the interactions between models and views. It provides the logical behavior and business logic of your applications. It chooses which view has to be rendered for a specific user request.

Generally speaking, since controllers provide the main application entry point, this means that they control how applications should respond to user requests.

Unit tests

The main goal of unit tests is to validate business logic. Normally, unit tests are put into their own external unit test projects. Multiple test frameworks are available for you to use, with the main ones being xUnit, NUnit, and MSTest.

As we mentioned previously, since everything is completely decoupled when using the MVC pattern, you can test your controllers at any point independently from the other parts of your applications by using unit tests.

WOW! eBook
www.wowebook.org

Integration tests

The end-to-end validation of application functionalities is done via integration tests.

Integration tests check that everything is working as expected from an application user point of view. Therefore, controllers and their corresponding views are tested together. Like unit tests, integration tests are normally put into their own testing projects and you can use a variety of testing frameworks (xUnit, NUnit, or MSTest). You may, however, also need to use a web server automation toolkit for this type of test, an example of which is Selenium. It must be noted that there is a thin line between integration tests and functional tests, which is a term that other developers use interchangeably, but nonetheless, they are different. Functional tests are more involved than integration tests in the sense that they are used for end-to-end testing and cover the functionality in an application. Integration tests are mainly there to see how certain components actually work with different components. In other words, how do they integrate?

Now that we have looked at the general and high-level picture of what an MVC application consists of, let's get some hands-on experience working with it. The best place to start is with views. This is what our users will be seeing; after all, we all know the term *customers first*, right? Our users may actually prefer to browse our application through a mobile phone instead of using a desktop PC. How are we going to know which device is accessing our application? How do we make our views responsive so that we can cater for different sized browser screens? We'll answer these questions in the next section.

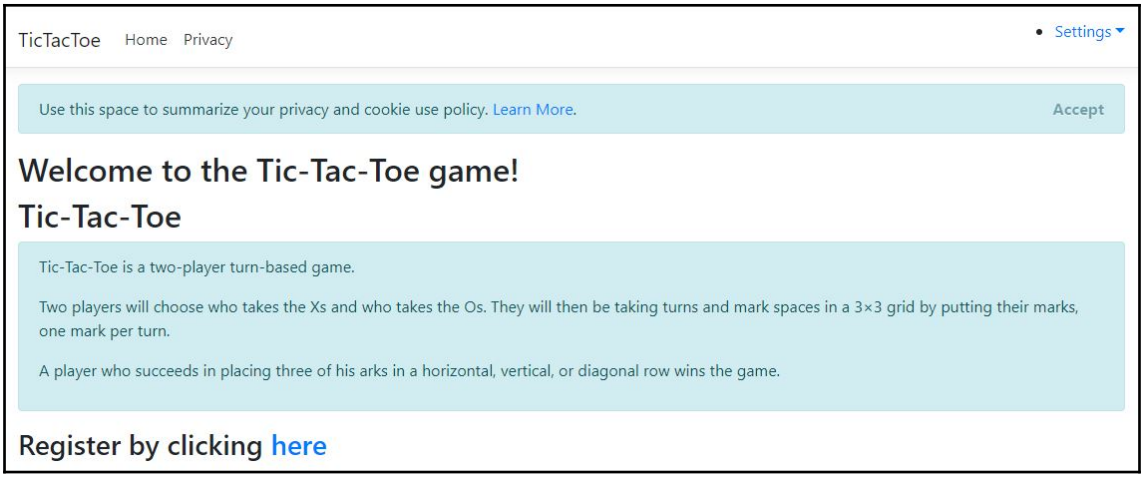
Creating dedicated layouts for multiple devices

Modern web applications use web page layouts to provide a consistent and coherent style. It is good practice to use HTML in combination with CSS to define this layout. In ASP.NET Core 3, the common web page layout definition is centralized on a layout page.

The layout page, usually called `_Layout.cshtml`, includes all the common user interface elements, such as the header, the menu, the sidebar, and the footer. Furthermore, common CSS and JavaScript files are referenced in the layout page so that they can be used throughout your entire application. This allows you to reduce code in your views, thus helping you to apply the **Don't Repeat Yourself** (**DRY**) principle.

WOW! eBook
www.wowebook.org

We have been using a layout page since the very early versions of the Tic-Tac-Toe demo application, that is, when we added it for the first time in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, to give our application a modern look, as you can see here:



Up until this point, our application has been more or less set up for the default browser, which is the PC. Is there any way we can differentiate our code based on respective devices? We'll look at the steps to achieve this in the next section.

The layout page in more detail

In this section, we will take a look at the layout page in more detail to understand what it is and how to take advantage of its features so that we can create dedicated layouts for multiple devices with different form factors (PCs, mobile phones, tablets, and more).

In `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, we added a layout page called `_Layout.cshtml` within the `Views/Shared` folder. When opening this page and analyzing its content, you can see that it contains common elements that are applicable to all the pages within your application (header, menu, footer, CSS, JavaScript, and more):

WOW! eBook
www.wowebook.org

```
_Layout.cshtml
1   <!DOCTYPE html>
2   <html>
3   <head>...</head>
20  <body>
21      <header>
22          <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
23              <div class="container">
24                  <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">TicTacToe</a>
25                  <button class="navbar-toggler">...</button>
29                  <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">...</div>
39              </div>
40          </nav>
41      </header>
42      <div class="container">
43          <partial name="_CookieConsentPartial" />
44          <main role="main" class="pb-3">
45              @RenderBody()
46          </main>
47      </div>
48
49      <footer class="border-top footer text-muted">
50          <div class="container">
51              &copy; 2019 - TicTacToe - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
52          </div>
53      </footer>
54
55      <environment>...</environment>
59      <environment>...</environment>
73      <script src="~/js/site.js" asp-append-version="true"></script>
74
75      @RenderSection("Scripts", required: false)
76  </body>
77  </html>
78
```

The common head section within the layout page contains CSS links but also **search engine optimization** (**SEO**) tags such as title, description, and keywords. As you have already seen, ASP.NET Core 3 provides a neat feature that allows you to include environment-specific content automatically via environment tags (development, staging, production, and more).

Bootstrap has become a quasi-standard for rendering menu and navigation bar components, which is why we have also used it for the Tic-Tac-Toe application.

It is good practice to put common JavaScript files at the bottom of your layout page; they can also be included depending on ASP.NET Core `environment` tags.

WOW! eBook
www.wowebook.org

We will use the `Views/_ViewStart.cshtml` file to define the layout page for all our pages in a central place. Alternatively, if you want to set a specific layout page manually, you can set it at the top of your page:

```
@{
    Layout = "_Layout";
}
```

To structure your layout pages, you can define sections so that you can organize where certain page elements, including common script sections, should be placed. An example is the script section that can be seen within the layout page, which we added in one of the first examples of the Tic-Tac-Toe application. By default, it has been put at the bottom of the page, which was done by adding a dedicated meta tag:

```
@RenderSection("Scripts", required: false)
```

You can also define sections in your views so that you can add files or client-side scripts. We have already done that in the context of the email confirmation view, where we added a section for calling the client-side JavaScript `EmailConfirmation` method:

```
@section Scripts{
    <script>
        $(document).ready(function () {
        EmailConfirmation('@ViewBag.Email');
        });
    </script>
}
```

We can also use **NuGet packages**. There is a package called `DeviceDetector.NET.NetCore` that is quite thorough in what it is able to detect, not only in terms of mobile devices but also other devices, including desktops, television sets, and even cars.

We can install and use the preceding package through the Package Manager Console by using the following command:

```
Install-Package DeviceDetector.NET.NetCore
```

For now, let's get practical and do the mobile detection functionality ourselves! Let's learn how to optimize the Tic-Tac-Toe application for mobile devices.

WOW! eBook
www.wowebook.org

Optimizing for mobile devices

Follow these steps take to make our Tic-Tac-Toe demo application more and more responsive for mobile devices, with the end goal of having a more convenient interface:

1. We want to change the display so that it's specifically for mobile devices. To do this, start Visual Studio 2019, go to the Solution Explorer, create a new folder called `Filters`, and add a new class called `DetectMobileFilter` that inherits from the `IActionFilter` interface. Then, we will create the `MobileCheck` and `MobileVersionCheck` **regular expressions** (**regex**), as follows:

```
static Regex MobileCheck = new
Regex(@"android|(android|bb\d+|meego).+mobile|avantgo|bada\/|blackb
erry|blazer
|compal|elaine|fennec|hiptop|iemobile|ip(hone|od)|iris|kindle
|lge |maemo|midp|mmp|mobile.+firefox|netfront|opera m(ob|in)i|palm(
os)?|phone|p(ixi|re)\/|plucker|pocket|psp|series(4|6)0|symbian
|treo|up\.
(browser|link)|vodafone|wap|windows (ce|phone)|xda|xiino",
RegexOptions.IgnoreCase | RegexOptions.Multiline |
RegexOptions.Compiled);


static Regex MobileVersionCheck = new
Regex(@"1207|6310|6590|3gso|4thp|50[1-6]i|770s|802s|a
wa|abac|ac(er|oo|s\-)|ai(ko|rn)|al(av|ca|co)|amoi|an(ex|ny|yw)
|aptu|ar(ch|go)|as(te|us)|attw|au(di|\-m|r |s
)|avan|be(ck|ll|nq)|bi(lb|rd)|bl(ac|az)|br(e|v)w|bumb|bw\-
(n|u)|c55\/|capi|ccwa|cdm\-|cell|chtm|cldc|cmd\-
|co(mp|nd)|craw|da(it|ll|ng)|dbte|dc\-
s|devi|dica|dmob|do(c|p)o|ds(12|\-
d)|el(49|ai)|em(l2|ul)|er(ic|k0)|esl8|ez([4-7]0|os|wa|ze)|fetc|fly(
\-|_)|g1 u|g560|gene|gf\-5|g\-
mo|go(\.w|od)|gr(ad|un)|haie|hcit|hd\-(m|p|t)|hei\-|hi(pt|ta)|hp(
i|ip)|hs\-c|ht(c(\-| |_|a|g|p|s|t)|tp)|hu(aw|tc)|i\-
(20|go|ma)|i230|iac( |\-
|\/)|ibro|idea|ig01|ikom|im1k|inno|ipaq|iris|ja(t|v)a|jbro|jemu|jig
s|kddi|keji|kgt( |\/)|klon|kpt |kwc\-|kyo(c|k)|le(no|xi)|lg(
g|\/(k|l|u)|50|54|\-[a-w])|libw|lynx|m1\-
w|m3ga|m50\/|ma(te|ui|xo)|mc(01|21|ca)|m\-
cr|me(rc|ri)|mi(o8|oa|ts)|mmef|mo(01|02|bi|de|do|t(\-|
|o|v)|zz)|mt(50|p1|v
)|mwbp|mywa|n10[0-2]|n20[2-3]|n30(0|2)|n50(0|2|5)|n7(0(0|1)|10)|ne(
(c|m)\-
|on|tf|wf|wg|wt)|nok(6|i)|nzph|o2im|op(ti|wv)|oran|owg1|p800|pan(a|
d|t)|pdxg|pg(13|\-
([1-8]|c))|phil|pire|pl(ay|uc)|pn\-2|po(ck|rt|se)|prox|psio|pt\-
```


WOW! eBook
www.wowebook.org

```
g|qa\-a|qc(07|12|21|32|60|\-[2-7]|i\-
)|qtek|r380|r600|raks|rim9|ro(ve|zo)|s55\/|sa(ge|ma|mm|ms|ny|va)|sc
(01|h\-|oo|p\-)|sdk\/|se(c(\-|0|1)|47|mc|nd|ri)|sgh\-|shar|sie(\-
|m)|sk\-0|sl(45|id)|sm(al|ar|b3|it|t5)|so(ft|ny)|sp(01|h\-|v\-|v
)|sy(01|mb)|t2(18|50)|t6(00|10|18)|ta(gt|lk)|tcl\-|tdg\-
|tel(i|m)|tim\-|t\-mo|to(pl|sh)|ts(70|m\-
|m3|m5)|tx\-9|up(\.b|g1|si)|utst|v400|v750|veri|vi(rg|te)|vk(40|5[0
-3]|\-v)|vm40|voda|vulc|vx(52|53|60|61|70|80|81|83|85|98)|w3c(\-|
)|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas\-|your|zeto|zte\-",
RegexOptions.IgnoreCase | RegexOptions.Multiline |
RegexOptions.Compiled);
```

2. Now, let's create a method that will check whether a user agent is mobile or not:

```
public static bool IsMobileUserAgent( ActionExecuted
 Context context)
{

    var userAgent = context.HttpContext.
    Request.Headers["User-Agent"].ToString();

    if (context.HttpContext != null && userAgent != null)
    {
        if (userAgent.Length < 4)
            return false;
        if (MobileCheck.IsMatch(userAgent) ||
        MobileVersionCheck.IsMatch(userAgent.
        Substring(0, 4)))
            return true;
    }
    return false;
}
```

3. Let's implement the `OnActionExecuted` method from `IActionFilter`, as follows:

```
public void OnActionExecuted(ActionExecutedContext context)
{
    var viewResult = (context.Result as ViewResult);
    if (viewResult == null)
        return;
    if (IsMobileUserAgent(context))
    {
        viewResult.ViewData["Layout"] =
        "~/Views/Shared/_LayoutMobile.cshtml";
    }
    else
    {
```


WOW! eBook
www.wowebook.org

```
                    viewResult.ViewData["Layout"] =
                    "~/Views/Shared/_Layout.cshtml";
                }
            }
```

4. Duplicate the existing `Views/Shared/_Layout.cshtml` file and rename the resulting copy `_LayoutMobile.cshtml`.
5. Update the home page index view, remove the existing layout definition, and display a different title, depending on the device, by adding two dedicated sections called `Desktop` and `Mobile`:

```
@{
ViewData["Title"] = "Home Page";
}
<div class="row">
<div class="col-lg-12">
@section Desktop {<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<div class="alert alert-info">
```

6. These sections will be made use of exclusively when a respective device is in use. This means that if a user is using a mobile phone for browsing, then only the `Mobile` section will appear on the screen.

> Note that you must also update all the other views of the application (`GameInvitation/GameInvitationConfirmation`, `GameInvitation/Index`, `Home/Index`, `UserRegistration/EmailConfirmation`, `UserRegistration/Index`) with the `section` tags from the preceding code for now:
> `@section Desktop{<h2>@Localizer["DesktopTitle"]</h2>}`
> `@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}`
> If you do not add them to your other views, you will get errors when you complete the steps that follow. However, this is only a temporary solution; later in this chapter, you will learn how to address this problem more effectively by using conditional statements.

6. Update the resource files. Here is an example of the English home page index resource file; you should also add the French translations:

Name	Value
DesktopTitle	Welcome to the Tic-Tac-Toe Desktop Game!
MobileTitle	Welcome to the Tic-Tac-Toe Mobile Game!

WOW! eBook
www.wowebook.org

7. Modify the `Views/Shared/_LayoutMobile.cshtml` file by replacing the `@RenderBody()` element with the following instructions; the `Desktop` section should be displayed and the `Mobile` section should be ignored:

```
@RenderSection("Desktop", required: false)
@{IgnoreSection("Mobile");}
@RenderBody()
```

8. Modify the `Views/Shared/_Layout.cshtml` file by replacing the `@RenderBody()` element with the following instructions; the `Mobile` section should be displayed and the `Desktop` section should be ignored:

```
@RenderSection("Mobile", required: false)
@{IgnoreSection("Desktop");}
@RenderBody()
```

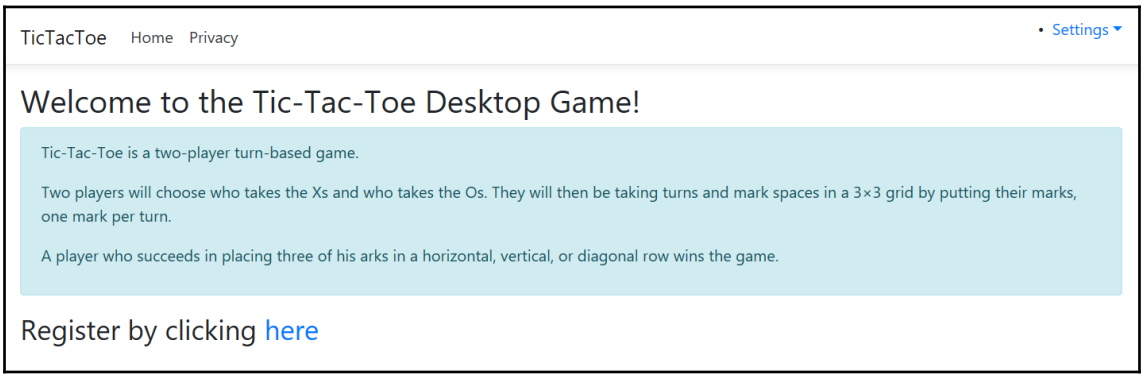
9. Go to the `Views/_ViewStart.cshtml` file and change the layout assignment for all your web pages to be able to use the layout definitions from the preceding code:

```
@{Layout = Convert.ToString(ViewData["Layout"]);}
```

10. Update the `Startup` class and add `DetectMobileFilter` to the MVC service registration as a parameter:

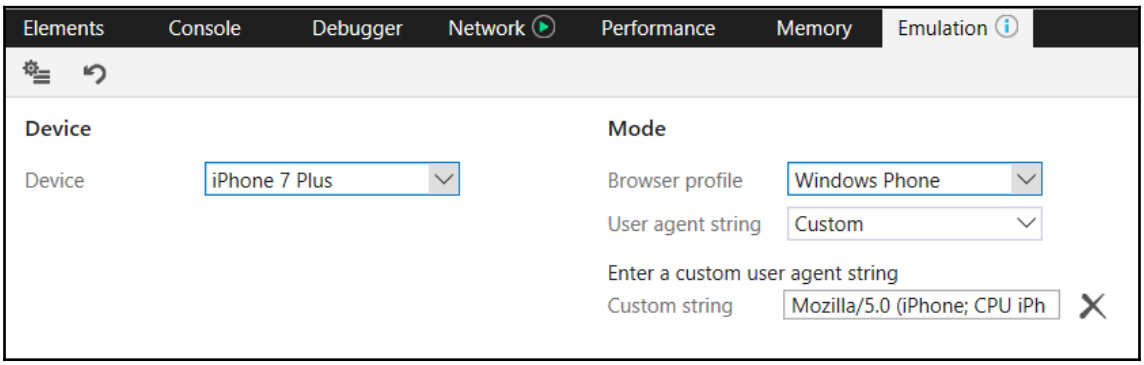
```
services.AddControllersWithViews(o =>
  o.Filters.Add(typeof(DetectMobileFilter)))
```

11. Start the Tic-Tac-Toe application normally in the Microsoft Edge browser. Note that the localized title already shows **Desktop** as the detected browser:

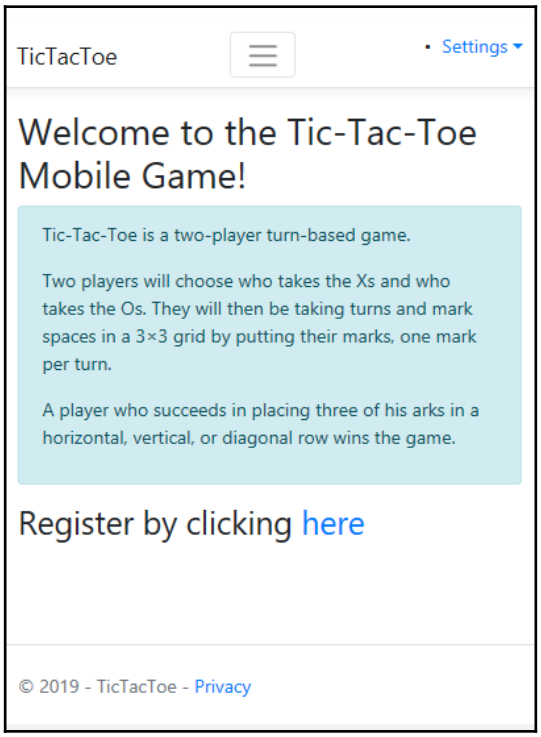


WOW! eBook
www.wowebook.org

Open the Developer Tools in Microsoft Edge by pressing *F12*, go to the **Emulation** tab, and select a mobile device:



Now, reload the Tic-Tac-Toe application; it will be displayed as if you had opened it on the emulated device:



WOW! eBook
www.wowebook.org

In this section, you have learned how to provide specific layouts for specific devices. Now, you are going to learn how to apply other advanced ASP.NET Core 3 MVC features for better productivity and better applications. But first, let's have a look at what's available to us in the **state management** between different requests, controllers, and views. We will have to deal with all of these at some point, even in the most basic of applications.

Understanding ASP.NET Core state management

An ASP.NET Core 3 web application is normally stateless. A brand new Razor web page is instantiated every time there is a request for the page from the server. As a result, any bit of data in the Razor page is effectively lost for each back and forth request.

To put things into perspective, we have already looked at a `UserRegistration` page, and if we inspect the browser with the developer tools, we will notice that when we fill the form and submit, our data is sent to the server but the data is not returned to the browser in the response headers.

This is the natural impediment of producing a meaningful and interactive application for the web, and we are lucky that ASP.NET Core 3 deals with this impediment for us with inbuilt features.

One of the ways we can make a decision on what feature functionality to use is by asking some of the following questions:

- Are we going to need to keep a lot of data between the requests?
- What are the chances that the user client will accept different kinds of cookies?
- Do we want the data to be stored by the server or client?
- Is the data in question delicate in that we need to keep it secure?
- Are we writing the application for user or client browsers that cannot afford high bandwidth usage?
- Exactly what kind of devices will be accessing our application? Do they have limitations?
- Will every application user need personalized data?
- How long do we need the application data to be persisted?
- Are we going to host our application in a distributed environment with multiple instances or a normal environment with a single instance?

Let's take a look at the options we have for managing states, along with their advantages.

WOW! eBook
www.wowebook.org

Client-state management options

For client-state management, the server doesn't need to store any data at all for any back and forth requests within the application. Let's take a look at these options in more detail.

Please note that we're referring to either the Razor pages or the user's device when we talk about the client.

Hidden fields

Hidden fields are a standard HTML functionality that requires no complex programming logic. Hidden fields have widespread support for most internet browsers. Since a hidden field is persisted and read from the Razor view page, no server resources are required.

ASP.NET Core 3 allows us to use hidden fields. Whatever you place in a hidden field will always be sent with the input data from other HTML elements in the form of a submission. This will be sent from where they were defined.

You can use `Hidden Field` to keep data on a `.cshtml` page and detect when data that's been stored in the hidden field has changed between postbacks.

It is recommended that you don't use a hidden field to keep sensitive data since the data can easily be revealed by right-clicking a web page and selecting **View Page Source**.

Cookies

Cookies are tiny bits of data stored by an application on a user's computer through a respective web browser. They can be used to keep customized client data, sessions by the user on the application, or application data. We can set a deliberate duration time on our cookies from milliseconds to minutes, hours, days, and even much longer, depending on how long you want to persist the data.

We will have a look at one of the most important usages for cookies in collaboration with ASP.NET Core Identity when we deal with authentication later in `Chapter 10`, *Securing ASP.NET Core 3 Applications*.

WOW! eBook
www.wowebook.org

We will deal with issues of security from `Chapter 10`, *Securing ASP.NET Core 3 Applications*, onward, but it is worth noting that cookies can present a vulnerability point to your application as they are often the targets of hackers. The best advice is to never store valuable information in your cookies but in tokens, which you use to locate your valuable data.

Query string

Sometimes, we may notice a key/value pair embedded in a `url` after a question mark tag, `:?`. This is an indication of a query string in use. Query strings come in handy when navigating between web pages and you have some data that you need to pass over to the next page. An example could be passing a game session `id` or other parameters for a game, which we could do for our demo application:

```
https://example.net/gamesessions?id=002e6431-3eb5-4d98-b3d9-3263490ce7c0
```

We must remember to always keep our `url` length relatively short, even though we have up to 2,048 characters as a maximum for modern browsers.

Query strings are a great tool, but there are scenarios where we shouldn't use them. We'll talk about this in more detail in the following subsection.

Query string usage

Query strings should be used when we need to request data from a web server using the `GET` method. Note that they cannot be used to send data to the web server using the `POST` method.

We will learn more about the `GET` and `POST` **Hypertext Transfer Protocol** (**HTTP**) methods in the next chapter, that is, `Chapter 8`, *Creating Web API Applications*.

Information that is passed in a query string is susceptible and can be tampered with by hackers. Please make sure you're not using query strings to pass important data. It must also be noted that an unsuspecting user can bookmark the URL or send the URL to other users, thereby passing that sensitive data in the URL along with it.

Query strings, cookies, and hidden fields are examples of client-side state management options, but most web applications will need to manage the state from the server side. The next section looks at the options you have for managing the state of the application from the server.

WOW! eBook
www.wowebook.org

Server-based state management options

ASP.NET Core 3 offers a variety of ways to maintain state information on the server, instead of saving data on the client. When the state is managed from the server side, there is a reduction in server-client calls. This could also prove to be expensive in terms of resources. The following sections will describe two server-based state management features: application state and session state.

Application state

Application state was used in previous frameworks prior to ASP.NET Core, for example, in ASP.NET MVC 4. It has now transitioned to being known as app state in ASP.NET Core 3. In its simplest terms, it is just a representation of the state of an application at a snapshot in time. When we refer to an application state, it will be the same for all the application users at the specified time. This allows us to keep data that remains constant across all the client users.

When you have this kind of data in a system that needs to be accessed across sessions and only changes every once in a while, it is advised to use caching. ASP.NET Core 3 prefers the usage of caching in favor of the application state.

Caching in ASP.NET Core 3 is achieved by using `IMemoryCache` and adding `services.AddMemoryCache()` to the `ConfigureServices` method, but it's out of the scope of this book. You can find more information at the following link: `https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-3.0`.

Session state

Session state, as opposed to application state, keeps user-specific data for the duration that they are using the application. ASP.NET Core 3 makes use of the session middleware, which is configured in the `ConfigureServices` method in the `Startup` class by adding `services.AddSession()`.

Before using the session object across your application, you need to add `app.UseSession()` to the `Configure` method in the `Startup` class.

By doing this, you can access the session object through `HttpContext.Session`, where you can utilize its methods to get or set your session variables or properties.

WOW! eBook
www.wowebook.org

We have already mentioned that the session state is a server-based state management option, and that means everything you set will be handled by the server. Note that you can actually store your session data in a cookie as an option in the `AddSession(option)` method, like so:

```
services.AddSession(options => {options.Cookie.Name =
"yourCustomSessionName";});
```

We are going to make use of session state in our game sessions later in this book by using the following code:

```
services.AddSession(o =>
{
    o.IdleTimeout = TimeSpan.FromMinutes(30);
});
```

The preceding code snippet is adding a session with a 30-minute timeout to the service collection in the `ConfigureServices` method. To be able to use this session, we will need to configure `app.UseSession()`, which can be found in the `Configure` method in the `Startup` class. More about sessions will be covered when we configure the game sessions in `Chapter 8`, *Creating Web API applications*, specifically in the *Building RPC-style web APIs* section.

Now that we have an idea of state management, let's have a look at the different types of views that we can make use of to give our application users a proper **user interface** (**UI**) and a good **user experience** (**UX**).

Using view pages, partial views, View Components, and Tag Helpers

ASP.NET Core 3 and Razor, when coupled with Visual Studio 2019, provide several functionalities that you can use to create your MVC views. In this section, you will learn how those functionalities can help you be more productive. You can, for instance, create views by using Visual Studio 2019's integrated scaffolding features, which you have already used in previous chapters multiple times. This allows you to automatically generate the following types of views:

- View pages
- Partial views

WOW! eBook
www.wowebook.org

Apart from view pages and partial views, sometimes, you need a bit more advanced functionality, which can be achieved using View Components and Tag Helpers.

Would you like to understand what they are and how to use Visual Studio 2019 to work with them efficiently? Stay focused – we are going to explain everything in detail in this section.

Using view pages

View pages are used to render results based on actions and to give responses to HTTP requests. In an MVC approach, they define and encapsulate the visible part of your applications – the presentation layer. Furthermore, they use the `.cshtml` file extension and are stored in the `Views` folder of the application by default.

Visual Studio 2019's scaffolding features provide different view page templates, as you can see here:

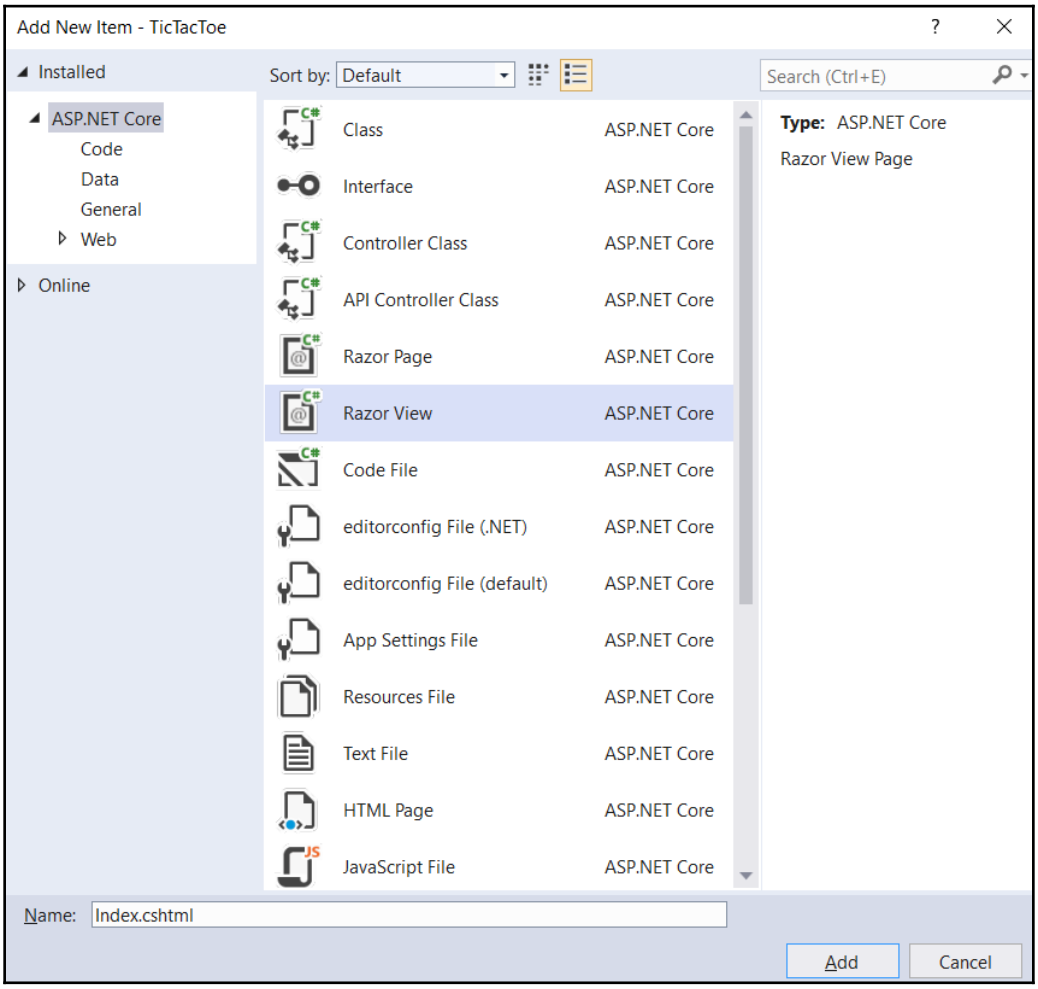
- **Create**: Generates a form for inserting data
- **Edit**: Generates a form for updating data
- **Delete**: Generates a form for displaying a record with a button to confirm deletion of the record data
- **Details**: Generates a form for displaying a record with two buttons, one for editing the form and another for deleting the displayed record page
- **List**: Generates an HTML table that shows a list of objects
- **Empty**: Generates an empty page without using any models

If you don't want to use Visual Studio 2019 to generate your page views, you can implement them manually by adding them to the `Views` folder yourself. In this case, it is advised that you respect the MVC conventions. So, add them in a corresponding subfolder while matching the action name. This helps the ASP.NET engine find your manually created views.

WOW! eBook
www.wowebook.org

Let's create the **Leaderboard** for the Tic-Tac-Toe game and see all of this in action:

1. Open the Solution Explorer, go to the **Views** folder, and create a new subfolder called `Leaderboard`. Then, right-click on the folder, select **Add | New Item | Razor View** page from the wizard, and click on the **Add** button:



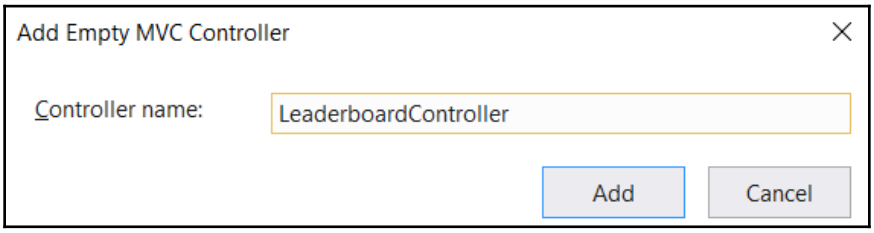
2. Open the created file and clear its content and associate the **Leaderboard** view with the **User Model** by adding the following instruction to the top of the page: `@model IEnumerable<UserModel>`.
3. It is good practice to set its title variable so that it's displayed in the SEO tags: `@{ViewData["Title"] = "Index";}`.

WOW! eBook
www.wowebook.org

4. Add new two sections, `Desktop` and `Mobile`, by using the `@section` meta tag. Then, add the last updated time by using the `@()` meta tag:

```
<div class="row">
        <div class="col-lg-12">
            @section Desktop {<h2>
                    @Localizer["DesktopTitle"] (
            Last updated @(System.DateTime.Now))
        </h2>}
            @section Mobile {<h2>
                    @Localizer["MobileTitle"] (
            Last updated @(System.DateTime.Now))
        </h2>}
        </div>
    </div>
```

5. Add the English and French resource files for the **Leaderboard** view and define localizations for the `DesktopTitle` and `MobileTitle`.
6. Right-click on the `Controllers` folder, select **Add | Controller**, select **MVC Controller - Empty**, and click on the **Add** button. Name it `LeaderboardController`:



7. The following code snippet will be auto-generated:

```
public class LeaderboardController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

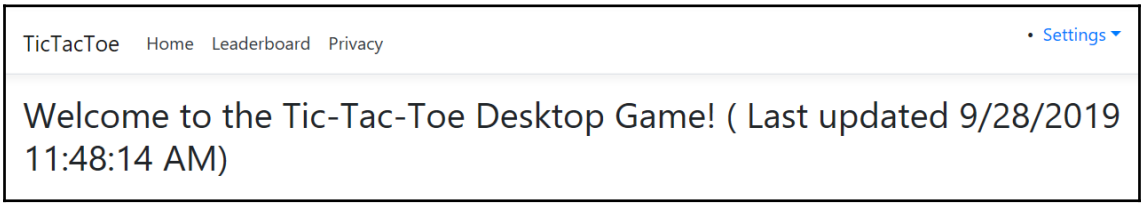

WOW! eBook
www.wowebook.org

Note that Razor matches views with actions with `<actionname>.cshtml` or `<actionname>.<culture>.cshtml` in the `Views/<controllername>` folder.

8. Update the `_Layout.cshtml` and `_LayoutMobile.cshtml` files in the `Views/Shared` folder and add an ASP.NET Tag Helper for calling the new **Leaderboard** view within the `navbar` menu, just after the `Home` element:

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-
    controller="Leaderboard" asp-action="Index">Leaderboard</a>
</li>
```

9. Start the application and display the new **Leaderboard** view:



Now that we have seen the basics, let's look at some more advanced techniques when using Razor, such as code blocks, control structures, and conditional statements. Code blocks, `@{}`, are used for setting or calculating variables and formatting data. You have already used them in the `_ViewStart.cshtml` file to define which specific layout page should be used:

```
@{
Layout = Convert.ToString(ViewData["Layout"]);
}
```

Control structures provide everything that's necessary for working with loops. You could use `@for`, `@foreach`, `@while`, or `@do` for repeating elements, for example. They act exactly the same as their C# equivalents. Let's use them to implement the **Leaderboard** view:

1. Add a new HTML table to the **Leaderboard** view while using the aforementioned control structures:

```
<table class="table table-striped">
    <thead> <tr>
            <th>Name</th>
            <th>Email</th>
```


WOW! eBook
www.wowebook.org

```
                <th>Score</th> </tr>
        </thead>
        <tbody>
            @foreach (var user in Model)
            { <tr>
                    <td>@user.FirstName @user.LastName</td>
                    <td>@user.Email</td>
                    <td>@user.Score.ToString()</td>  </tr>
            }
        </tbody>
    </table>
```

2. Add a new `GetTopUsers` method to the `IUserService` interface for retrieving the top users that will be displayed within the **Leaderboard** view:

```
Task<IEnumerable<UserModel>> GetTopUsers(int numberOfUsers);
```

3. Implement the new `GetTopUsers` method within `UserService`:

```
public Task<IEnumerable<UserModel>> GetTopUsers(int
 numberOfUsers)
{
    return Task.Run(() =>
    (IEnumerable<UserModel>)_userStore.OrderBy(x =>
    x.Score).Take(numberOfUsers).ToList());
}
```

4. Update the leaderboard controller so that you can call the new method:

```
private IUserService _userService;
public LeaderboardController(IUserService userService)
{
    _userService = userService;
}
public async Task<IActionResult> Index()
{
    var users = await _userService.GetTopUsers(10);
    return View(users);
}
```


WOW! eBook
www.wowebook.org

5. Press *F5* and start the application, register multiple users, and display the leaderboard:

TicTacToe Home Leaderboard Privacy • Settings

Welcome to the Tic-Tac-Toe Desktop Game! (Last updated 9/28/2019 12:23:16 PM)

Name	Email	Score
Jane Doe	jane.doe@gmail.com	0
Amara Taoloka	amara_t@gmail.com	0
Kenneth Fukizi	ken.fukizi@gmail.com	0
John Doe	john.doe@gmail.com	0

Conditional statements such as `@if`, `@else if`, `@else`, and `@switch` allow you to render elements conditionally. They also work exactly the same as their C# counterparts.

As we mentioned previously, you need to define the `Desktop` and `Mobile` sections in all of your views, that is, `@section Desktop { }` and `@section Mobile { }`.

For example, if you remove them temporarily from the **Leaderboard Index View** and try to display it while the `ASPNETCORE_ENVIRONMENT` variable is set to `'Development'` so that the Developer Exception page is activated, you will get the following error message:

WOW! eBook
www.wowebook.org

An unhandled exception occurred while processing the request.

InvalidOperationException: The layout page '/Views/Shared/_Layout.cshtml' cannot find the section 'Mobile' in the content page '/Views/LeaderBoard/Index.cshtml'.

Microsoft.AspNetCore.Mvc.Razor.RazorPage.IgnoreSection(string sectionName)

Stack Query Cookies Headers Routing

InvalidOperationException: The layout page '/Views/Shared/_Layout.cshtml' cannot find the section 'Mobile' in the content page '/Views/LeaderBoard/Index.cshtml'.

Microsoft.AspNetCore.Mvc.Razor.RazorPage.IgnoreSection(string sectionName)
AspNetCore.Views_Shared__Layout.<ExecuteAsync>b__50_1() in **_Layout.cshtml**
+ 81. @{IgnoreSection("Mobile");}
Microsoft.AspNetCore.Razor.Runtime.TagHelpers.TagHelperExecutionContext.SetOutputContentAsync()
AspNetCore.Views_Shared__Layout.ExecuteAsync()
Microsoft.AspNetCore.Mvc.Razor.RazorView.RenderPageCoreAsync(IRazorPage page, ViewContext context)

This has happened because we changed the `Layout` and `Mobile` layout pages for the application and used the `IgnoreSection` instruction. Unfortunately, sections must always be declared when using `IgnoreSection` instructions.

But now that you know that conditional statements exist, you can already see a better solution, right? Yes, exactly; we have to wrap the `IgnoreSection` instruction with a conditional `if` statement within the two layout pages.

Here's how you need to update the layout page using the `IsSectionDefined` method:

```
@RenderSection("Desktop", required: false)
@if(IsSectionDefined("Mobile")){IgnoreSection("Mobile");}
@RenderBody()
```

Here's how you need to update the `Mobile` layout page:

```
@RenderSection("Mobile", required: false)
@if(IsSectionDefined("Desktop")){IgnoreSection("Desktop");}
@RenderBody()
```

If you start the application, you will see that everything is now working as expected, but this time with a much cleaner, more elegant, and easier-to-understand solution. This is because we're using the built-in functionalities of ASP.NET Core 3 and Razor.

WOW! eBook
www.wowebook.org

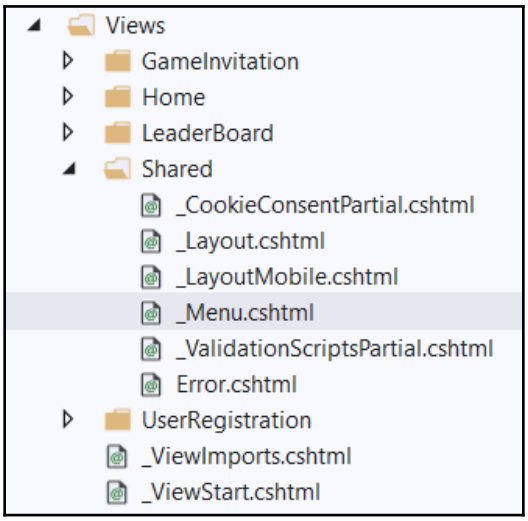
Using partial views

So far, we've learned how to create view pages using Razor, but sometimes, we have to repeat elements within all or some of our view pages. Wouldn't it be helpful if we could create reusable components within our views? Unsurprisingly, ASP.NET Core 3 implements this feature by default by providing so-called partial views.

Partial views are rendered within calling view pages. Like standard view pages, they also have the `.cshtml` file extension. We can define them once and then use them within all our view pages. What a great way to optimize our code by reducing code duplication, which leads to better quality and less maintenance!

Let's look at how we can benefit from this right now. To do this, we will be optimizing the Layout and Mobile layout pages so that they only use a single menu:

1. Go to the **Views/Shared** folder and add a new MVC view page called `_Menu.cshtml`. It will be used as the menu partial view:



2. Copy the `nav bar` from one of the layout pages and paste it into the menu partial view:

```
<div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-
reverse">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
```


WOW! eBook
www.wowebook.org

```
                <a class="nav-link text-dark" asp-area="" asp-
                    controller="Home" asp-action="Index">Home</a>
            </li>

            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-
                 controller="Leaderboard" asp-
                 action="Index">Leaderboard</a>
            </li>

            <li class="nav-item">
                <a class="nav-link text-dark" asp-area="" asp-
                  controller="Home" asp-action="Privacy">Privacy</a>
            </li>
        </ul>
    </div>
```

3. Replace `nav bar` with `<partial name="_Menu" />` in both layout pages.
4. Start the application and validate that everything is still working. You shouldn't see any differences, but that is a good thing; you have encapsulated and centralized the menu in a partial view now.

Using View Components

So far, you've learned how to create reusable components by using partial views, which can be called from any view pages within your applications, and applied this concept to the top menu of the Tic-Tac-Toe application. But sometimes, even this feature is not enough. Sometimes, you need something more powerful – something more flexible – that you can use throughout your whole web application and maybe even for multiple web applications. That is where View Components come into play.

View Components are used for complex use cases that require some code running on the server (for example, login panel, tag cloud, and shopping cart), where partial views are too limited to be used, and where you need to be able to test functionalities extensively. We are going to add a View Component for managing game sessions in the following example. You will see that it is very similar to a standard controller implementation:

1. Add a new model called `TurnModel` to the `Models` folder:

```
public class TurnModel
    {
        public Guid Id { get; set; }
        public Guid UserId { get; set; }
        public UserModel User { get; set; }
```


WOW! eBook
www.wowebook.org

```
        public int X { get; set; }
        public int Y { get; set; }
    }
```

2. Add a new model called `GameSessionModel` to the `Models` folder:

```
public class GameSessionModel
    {
        public Guid Id { get; set; }
        public Guid UserId1 { get; set; }
        public Guid UserId2 { get; set; }
        public UserModel User1 { get; set; }
        public UserModel User2 { get; set; }
        public IEnumerable<TurnModel> Turns { get; set; }
        public UserModel Winner { get; set; }
        public UserModel ActiveUser { get; set; }
        public Guid WinnerId { get; set; }
        public Guid ActiveUserId { get; set; }
        public bool TurnFinished { get; set; }
    }
```

3. Add a new service called `GameSessionService` to the `Services` folder, implement it, and extract the `IGameSessionService` interface:

```
public class GameSessionService : IGameSessionService
    {
        private static ConcurrentBag<GameSessionModel> _sessions;
        static GameSessionService()
        {
            _sessions = new ConcurrentBag<GameSessionModel>();
        }
        public Task<GameSessionModel> GetGameSession(Guid
         gameSessionId)
        {
            return Task.Run(() => _sessions.FirstOrDefault(
            x => x.Id == gameSessionId));
        }
    }
```

4. Register `GameSessionService` within the `Startup` class, like you did with all the other services:

```
services.AddSingleton<IGameSessionService, GameSessionService>();
```


WOW! eBook
www.wowebook.org

5. Go to the Solution Explorer, create a new folder called `Components`, and add a new class called `GameSessionViewComponent.cs` to it:

```
[ViewComponent(Name = "GameSession")]
    public class GameSessionViewComponent : ViewComponent
    {
        IGameSessionService _gameSessionService;
        public GameSessionViewComponent(IGameSessionService
         gameSessionService)
        {
            _gameSessionService = gameSessionService;
        }
        public async Task<IViewComponentResult> InvokeAsync(Guid
         gameSessionId)
        {
            var session = await _gameSessionService.
             GetGameSession(gameSessionId);
            return View(session);
        }
    }
```

6. Go to the Solution Explorer and create a new folder called `Components` within the `Views/Shared` folder. Within this folder, create a new folder called `GameSession` for `GameSessionViewComponent`. Then, manually add a new view called `default.cshtml`:

```
@model TicTacToe.Models.GameSessionModel
@{     var email = Context.Session.GetString("email"); }
@if (Model.ActiveUser?.Email == email)
{<table>
 @for (int rows = 0; rows < 3; rows++)  {<tr style="height:150px;">
   @for (int columns = 0; columns < 3; columns++)
   {<td style="width:150px; border:1px solid #808080">
   @{var position = Model.Turns?.FirstOrDefault(turn => turn.X ==
     columns && turn.Y == rows);
    if (position != null) { if (position.User?.Email == "Player1")
     {<i class="glyphicon glyphicon-unchecked"
      style="width:100%;height:100%"></i> }
     else{<i class="glyphicon glyphicon-remove-circle"
      style="width:100%;height:100%"></i> }
    } else{ <a asp-action="SetPosition"
      asp-controller="GameSession"
     asp-route-id="@Model.Id" asp-route-email="@email"
     class="btn btn-default" style="width:150px;
      min-height:150px;">
          </a> } } </td> } </tr> }
    </table>
```


WOW! eBook
www.wowebook.org

```
}else{
 <div class="alert">
   <i class="glyphicon glyphicon-alert">Please wait until the other
user has finished his turn.</i> </div> }
```

This is the view that will tell you to wait for your turn if you're not the active user; otherwise, it will give you a table where you can play the `TicTacToe` game.

We advise using the following syntax to put all partial views for your View Components in their corresponding folders: `Views\Shared\Components\<ViewComponentName>\<ViewName>`.

7. Update the `_ViewImports.cshtml` file to use the View Component by using the `@addTagHelper *, TicTacToe` command.
8. Create a new folder called `GameSession` within the `Views` folder. Then, add a new view called `Index` for the `Desktop` section, as follows:

```
@model TicTacToe.Models.GameSessionModel
@section Desktop
{<h1>Game Session @Model.Id</h1>
    <h2>Started at @(DateTime.Now.ToShortTimeString())</h2>
    <div class="alert alert-info">
        <table class="table">
         <tr>
           <td>User 1:</td>
           <td>@Model.User1?.Email (<i class="glyphicon
             glyphicon-unchecked"></i>) </td>
          </tr>
          <tr>
            <td>User 2:</td>
            <td>@Model.User2?.Email (<i class=" glyphicon
             glyphicon-remove-circle"></i></td>
        </tr>
    </table>
</div>}
```


WOW! eBook
www.wowebook.org

Now, do the same for the `Mobile` section, as follows:

```
@section Mobile{
    <h1>Game Session @Model.Id</h1>
    <h2>Started at @(DateTime.Now.ToShortTimeString())</h2>
    User 1: @Model.User1?.Email <i class="glyphicon glyphicon-
        unchecked"></i><br />
    User 2: @Model.User2?.Email (<i class="glyphicon glyphicon-
        remove-circle"></i>)
}
<vc:game-session game-session-id="@Model.Id"></vc:game-session>
```

9. Add a public constructor to `GameSessionService` so that you can get an instance of the user service:

```
private IUserService _UserService;
public GameSessionService(IUserService userService)
{
    _UserService = userService;
}
```

10. Add a method to `GameSessionService` for creating game sessions and update the game session service interface:

```
public async Task<GameSessionModel> CreateGameSession( Guid
invitationId, string invitedByEmail, string invitedPlayerEmail)
        {
            var invitedBy =
            await _UserService.GetUserByEmail(invitedByEmail);
            var invitedPlayer =
            await _UserService.GetUserByEmail(invitedPlayerEmail);
            GameSessionModel session = new GameSessionModel
            {
                User1 = invitedBy,
                User2 = invitedPlayer,
                Id = invitationId,
                ActiveUser = invitedBy
            };
            _sessions.Add(session);
            return session;
        }
```

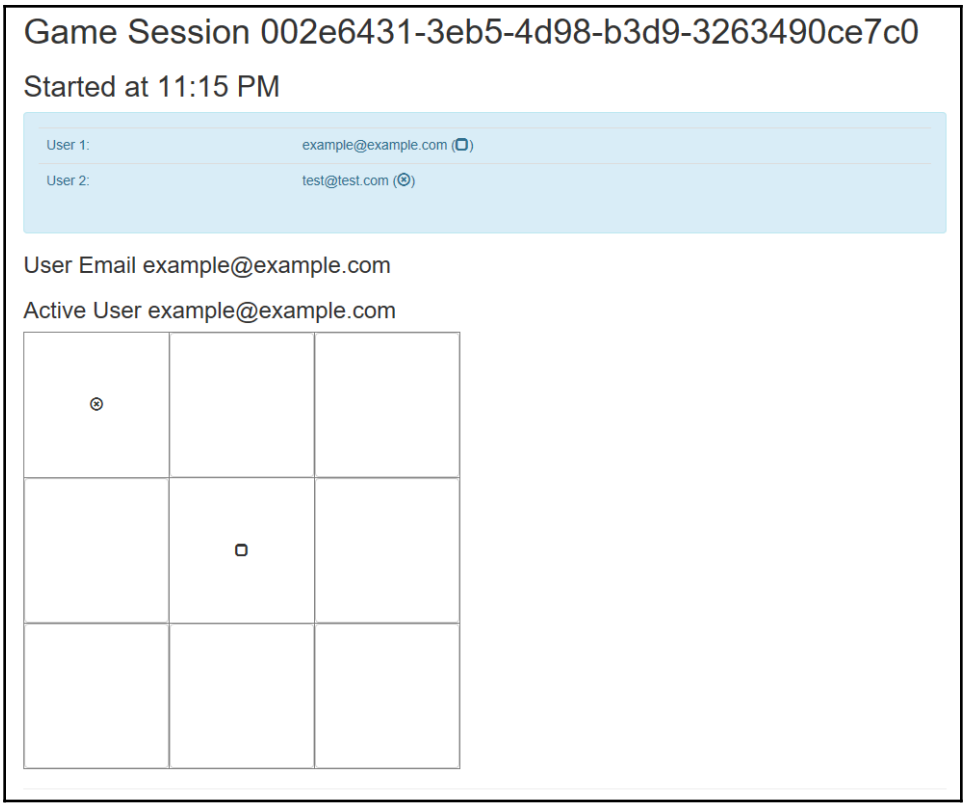

WOW! eBook
www.wowebook.org

11. Add a new controller called `GameSessionController` within the `Controllers` folder and implement a new `Index` method:

```
private IGameSessionService _gameSessionService;
public GameSessionController(IGameSessionService
gameSessionService)
{  _gameSessionService = gameSessionService;  }
public async Task<IActionResult> Index(Guid id)
{
    var session = await _gameSessionService.
        GetGameSession(id);
    if (session == null)
    {
        var gameInvitationService =
        Request.HttpContext.
        RequestServices.GetService
        <IGameInvitationService>();
        var invitation = await gameInvitationService.
          Get(id);
        session = await _gameSessionService.
        CreateGameSession(
                    invitation.Id, invitation.InvitedBy,
                     invitation.EmailTo);
    }
    return View(session);
}
```

12. Start the application, register a new user, and invite another user to play a game. Wait for the new game session page to be displayed, as follows:

WOW! eBook
www.wowebook.org



In this section, we've learned how to implement an advanced feature called View Components. In the next section, we will take a look at another advanced and exciting feature called Tag Helpers. Stay focused.

Using Tag Helpers

Tag Helpers are a relatively new feature since ASP.NET Core 2+ and allow server-side code to be used when creating and rendering HTML elements. They can be compared to the well-known HTML helpers for rendering HTML content.

ASP.NET Core 3 provides many built-in Tag Helpers, such as `ImageTagHelper` and `LabelTagHelper`, that you can use within your applications. When creating your own Tag Helpers, you can target HTML elements based on an element name, an attribute name or a parent tag. Then, you can use standard HTML tags in your views while presentation logic written in C# is applied on the web server.

WOW! eBook
www.wowebook.org

Additionally, you can even create custom tags. You can use these within the Tic-Tac-Toe demo application. Let's learn how to create custom tags:

1. Open the Solution Explorer and create a new folder called `TagHelpers`. Then, add a new class called `GravatarTagHelper.cs` that implements the `TagHelper` base class.
2. Implement the `GravatarTagHelper.cs` class; it will be used to connect to the Gravatar online service for retrieving account photos for users. Let's start with the plumbing for the class:

```
[HtmlTargetElement("Gravatar")]
    public class GravatarTagHelper : TagHelper
    {
        private ILogger<GravatarTagHelper> _logger;
        public GravatarTagHelper(ILogger<GravatarTagHelper> logger)
        {
            _logger = logger;
        }
        public string Email { get; set; }
...
}
```

Now, we can implement the `Process` method, as follows:

```
public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            byte[] photo = null;
            if (CheckIsConnected())
            {
                photo = GetPhoto(Email);
            }
            else
            {
                photo = File.ReadAllBytes(Path.Combine(
                Directory.GetCurrentDirectory(),
                "wwwroot", "images", "no-photo.jpg"));
            }
            string base64String = Convert.ToBase64String(photo);
            output.TagName = "img";
            output.Attributes.SetAttribute("src",
            $"{base64String}");
        }
```


WOW! eBook
www.wowebook.org

The `Process` method will require a method to check for a connection, called `CheckIsConnected`, which can be implemented as follows:

```
private bool CheckIsConnected()
{
    try
    {
        using (var httpClient = new HttpClient())
        {
            var gravatarResponse = httpClient.GetAsync(
            "http://www.gravatar.com/avatar/").Result;
            return (gravatarResponse.IsSuccessStatusCode);
        }
    }
    catch (Exception ex)
    {
        _logger?.LogError($"Cannot check the gravatar
          service status: { ex} ");
        return false;
    }
}
```

We will also need a `GetPhoto` method, as follows:

```
private byte[] GetPhoto(string email)
        {
            var httpClient = new HttpClient();
            return httpClient.GetByteArrayAsync(
            new Uri($"http://www.gravatar.com/avatar/ {
              HashEmailForGravatar(email) }")).Result;
        }
```

Finally, we need a `HashEmailForGravatar` method, as follows:

```
private static string HashEmailForGravatar(string email)
        {
            var md5Hasher = MD5.Create();
            byte[] data = md5Hasher.ComputeHash(
            Encoding.ASCII.GetBytes(email.ToLower()));
            var stringBuilder = new StringBuilder();
            for (int i = 0; i < data.Length; i++)
            {
                stringBuilder.Append(data[i].ToString("x2"));
            }
            return stringBuilder.ToString();
        }
```


WOW! eBook
www.wowebook.org

3. Open the `Views/_ViewImports.cshtml` file and verify that the `addTagHelper` instruction exists. If it doesn't add it to the file by using the `@addTagHelper *, TicTacToe` command.
4. Update the `Index` method in the `GameInvitationController`, store the user's email, and display their name (first name and last name) in a session variable:

```
[HttpGet]
public async Task<IActionResult> Index(string email)
{
    var gameInvitationModel = new GameInvitationModel
    {
        InvitedBy = email,
        Id = Guid.NewGuid()
    };
    Request.HttpContext.Session.SetString("email", email);
    var user = await _userService.GetUserByEmail(email);
    Request.HttpContext.Session.SetString("displayName",
    $"{user.FirstName} {user.LastName}");
    return View(gameInvitationModel);
}
```

5. Add a new model called `AccountModel` to the `Models` folder:

```
public class AccountModel
 {
     public string Email { get; set; }
     public string DisplayName { get; set; }
 }
```

6. Add a new partial view called `_Account.cshtml` to the `Views/Shared` folder:

```
@model TicTacToe.Models.AccountModel
<li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown">
        <span class="glyphicon glyphicon-user"></span>
        <strong>@Model.DisplayName</strong>
        <span class="glyphicon glyphicon-chevron-down"></span>
    </a>
    <ul class="dropdown-menu" id="connected-dp">
        <li>
            <div class="navbar-login">
                <div class="row">
                    <div class="col-lg-4">
                        <p class="text-center">
                            <Gravatar email="@Model.Email">
                            </Gravatar>
                        </p>
```


WOW! eBook
www.wowebook.org

```
                        </div>
                        <div class="col-lg-8">
                            <p class="text-left"><strong>@Model.
                             DisplayName</strong></p>
                            <p class="text-left small">
                                <a asp-action="Index" asp-
                                controller="Account">
                                @Model.Email</a>
                            </p>
                        </div>
                    </div>
                </div>
            </li>
            <li class="divider"></li>
            <li>
                <div class="navbar-login navbar-login-session">
                    <div class="row">
                        <div class="col-lg-12">
                            <p>
                                <a href="#" class="btn btn-danger btn-
                                  block">Log off</a>
                            </p>
                        </div>
                    </div>
                </div>
            </li>
        </ul>
    </li>
```

7. Add a new CSS class to the `wwwroot/css/site.css` file:

```
#connected-dp {
             min-width: 350px;
             }
```

Note that you might need to empty your browser cache or force a refresh for the application so that you can update the `site.css` file within your browser.

WOW! eBook
www.wowebook.org

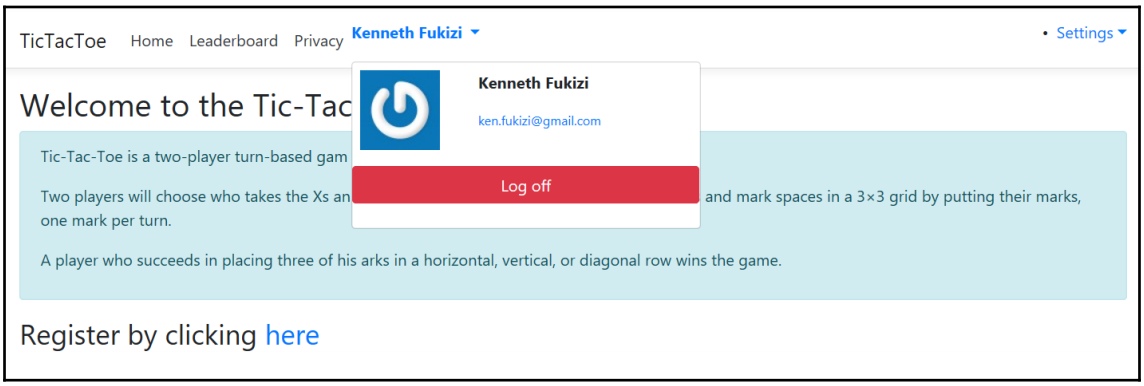
8. Update the menu partial view and retrieve the user display name and email at the top of the page:

```
@using Microsoft.AspNetCore.Http;
@{
    var email = Context.Session.GetString("email");
    var displayName = Context.Session.GetString("displayName");
}
```

9. Update the menu partial view and add the new account partial view that we created previously. This can be found after the privacy element in the menu:

```
<li>
    @if (!string.IsNullOrEmpty(email))
    {
        Html.RenderPartial("_Account",
        new TicTacToe.Models.AccountModel
        {
            Email = email,
            DisplayName = displayName
        });
    }
</li>
```

10. Create an account on Gravatar with your email and upload a photo. Start the Tic-Tac-Toe application and register with the same email. You should now see a new dropdown with a photo and display name in the top menu:



Note that you have to be online for this to work. If you want to test your code offline, you should put a photo in the `wwwroot/images` folder called `no-photo.jpg`; otherwise, you will get an error since no offline photo can be found.

WOW! eBook
www.wowebook.org

This should be easy to understand and easy to use, but when should we use View Components and when should we use Tag Helpers? The following simple rules should help you decide when to use either:

- View Components are used whenever we need templates for views, when we need to render a group of elements, and when we need to associate server code with it.
- Tag Helpers are used to append behavior to a single HTML element instead of a group of elements.

Our application is growing. For larger applications, it can become a nightmare to logically follow the application, especially if you are a new developer that's been placed on an existing project – it might take you some time to get used to the code base. Luckily, ASP.NET Core 3 allows us to compartmentalize similar functionality. We'll look at how to do this in the next section.

Dividing a web application into multiple areas

Sometimes, when working with larger web applications, it can be interesting to logically separate them into smaller functional units. Each unit can then have its own controllers, views, and models, which makes it easier to understand, manage, evolve, and maintain them over time.

ASP.NET Core 3 provides some simple mechanisms based on the folder's structure for dividing web applications into multiple functional units, also called `Areas`; for example, to separate the standard `Area` from the more advanced administration `Area` within your applications. The standard `Area` could even enable anonymous access on some pages while asking for authentication and authorization on others, whereas the administration `Area` would always require authentication and authorization on all pages.

The following conventions and restrictions apply to `Areas`:

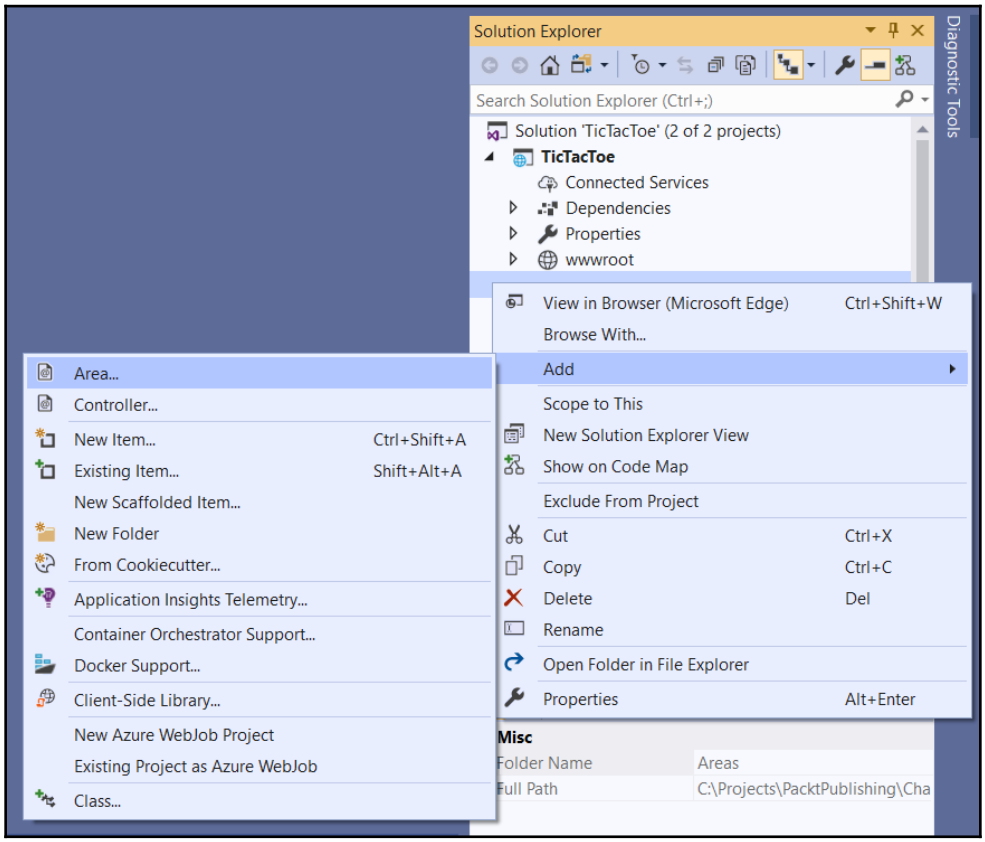
- An `Area` is a subdirectory in the `Areas` folder.
- An `Area` contains at least two subfolders: `Controllers` and `Views`.
- An `Area` may contain specific layout pages, as well as dedicated `_ViewImport.cshtml` and `_ViewStart.cshtml` files.

WOW! eBook
www.wowebook.org

- You have to register a specific route that enables `Areas` within its routing definition to be able to use `Areas` in your applications.
- It is recommended to use the following format for `Area` URLs: `http://<Host>/<AreaName>/<ControllerName>/<ActionName>`.
- The `asp-area` Tag Helper can be used for appending an `Area` to a URL.

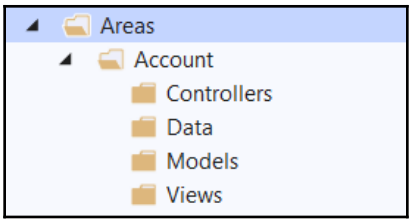
Let's look at how to create a specific administration `Area` for account management:

1. Open the Solution Explorer and create a new folder called **Areas**. Right-click on the folder, select **Add | Area...**, enter `Account` as the `Area` name, and click on the **Add** button:



2. Scaffolding will create a dedicated folder structure for the `Account Area`, as follows:

WOW! eBook
www.wowebook.org



3. Add a new route for `Areas` to the `UseEndpoints` declaration within the `Configure` method of the `Startup` class:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "
        {controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
    endpoints.MapAreaControllerRoute(
        name: "areas",
        areaName: "Account",
        pattern : "
        {area:exists}/{controller=Home}
        /{action=Index}/{id?}"
        );
});
```

4. Right-click on the `Controllers` folder within the `Account Area` and add a new controller called `HomeController`:

```
[Area("Account")]
    public class HomeController : Controller
    {
        private IUserService _userService;
        public HomeController(IUserService userService)
        {
            _userService = userService;
        }
        public async Task<IActionResult> Index()
        {
            var email = HttpContext.Session.GetString("email");
            var user = await _userService.GetUserByEmail(email);
            return View(user);
        }
    }
```


WOW! eBook
www.wowebook.org

5. Add a new folder called `Home` in the `Account/Views` folder. Then, add a view called `Index` in this new folder:

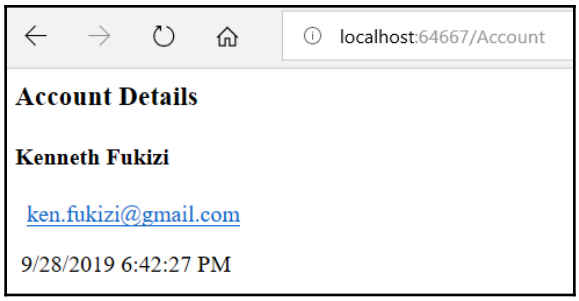
```
@model TicTacToe.Models.UserModel
<h3>Account Details</h3>
<div class="container">
    <div class="row">
        <div class="col-xs-12 col-sm-6 col-md-6">
            <div class="well well-sm">
                <div class="row">
                    <div class="col-sm-6 col-md-4">
                        <Gravatar email="@Model.Email"></Gravatar>
                    </div>
                    <div class="col-sm-6 col-md-8">
                        <h4>@($"{Model.FirstName}
                         {Model.LastName}")</h4>
                        <p>
                            <i class="glyphicon glyphicon
                            -envelope"></i> 
                            <a href="mailto:@Model.Email">@Model.
                             Email</a>
                        </p>
                        <p>
                            <i class="glyphicon glyphicon
                             -calendar">
                            </i> @Model.EmailConfirmationDate
                        </p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

6. Update the account partial view and add a link to display the preceding view (just after the existing log off link):

```
<a class="btn btn-default btn-block" asp-action="Index"
asp-controller="Account">View Details</a>
```

7. Start the application, register a user, and call the new `Area` by clicking on the **View Details** link on the dropdown:

WOW! eBook
www.wowebook.org



We will stop the implementation of the administration Area here and come back to it in `Chapter 10`, *Securing ASP.NET Core 3 Applications*, where you will learn how to secure access to it. For now, let's get a bit more advanced by looking at an exciting feature called the view engine. The more advanced we get, the more complex our codebase will become, and one of the best ways to ensure that we always get the intended functionality is to write unit tests and integration tests. We'll introduce these in the next section as well.

Applying advanced concepts such as view engines, unit tests, and integration tests

Now that we have seen all the basic features of ASP.NET Core 3 MVC, let's look at some of the more advanced features that can help you during your daily work as a developer. You will also learn how to use Visual Studio 2019 to test your applications and thus provide better quality for your users.

Using view engines

When ASP.NET Core 3 uses server-side code for rendering HTML, it uses a view engine. By default, when building standard views with their associated `.cshtml` files, we use the Razor view engine with the Razor syntax, for example.

By convention, this engine is able to work with views, which are located within the `Views` folder. Since it is built-in and it is the default engine, it is automatically bound to the HTTP request pipeline without us needing to do anything for it to work.

If we need to use Razor to render files that are located outside of the `Views` folder and don't come directly from the HTTP request pipeline, such as an email template, we cannot use the default Razor view engine. Instead, we need to define our own view engine and make it responsible for generating the HTML code.

WOW! eBook
www.wowebook.org

In the following example, we will explain how you can use Razor to render an email based on an email template that isn't coming from the HTTP request pipeline:

1. Open the Solution Explorer and create a new folder called `ViewEngines`. Then, add a new class called `EmailViewEngine.cs` that has the following constructor:

```
public class EmailViewEngine
    {
        private readonly IRazorViewEngine _viewEngine;
        private readonly ITempDataProvider _tempDataProvider;
        private readonly IServiceProvider _serviceProvider;
        public EmailViewEngine( IRazorViewEngine viewEngine,
         ITempDataProvider tempDataProvider, IServiceProvider
         serviceProvider)
        {
            _viewEngine = viewEngine;
            _tempDataProvider = tempDataProvider;
            _serviceProvider = serviceProvider;
        }
        ...
    }
```

Within the same `EmailViewEngine`, let's create a `FindView` method, as follows:

```
 private IView FindView(ActionContext actionContext, string
viewName)
 {
   var getViewResult = _viewEngine.GetView(executingFilePath:
     null, viewPath: viewName, isMainPage: true);
   if (getViewResult.Success)
       return getViewResult.View;
   var findViewResult = _viewEngine.FindView(actionContext,
    viewName, isMainPage: true);
   if (findViewResult.Success)
       return findViewResult.View;
   var searchedLocations = getViewResult.
    SearchedLocations.Concat(findViewResult.SearchedLocations);
   var errorMessage = string.Join
                ( Environment.NewLine, new[] { $"Unable to
                  find view '{viewName}'. The following locations
                   were searched:" }.Concat(searchedLocations));
                throw new InvalidOperationException(errorMessage);
 }
```


WOW! eBook
www.wowebook.org

Let's create a `GetActionContext` method in the same `EmailViewEngine` class:

```
private ActionContext GetActionContext()
        {
            var httpContext = new DefaultHttpContext
            {
                RequestServices = _serviceProvider
            };
            return new ActionContext(httpContext, new RouteData(),
            new ActionDescriptor());
        }
```

We will use the preceding method in the following `RenderEmailToString` method, as follows:

```
public async Task<string> RenderEmailToString<TModel>(string
viewName, TModel model)
        {
            var actionContext = GetActionContext();
            var view = FindView(actionContext, viewName);
            if (view == null)
                throw new InvalidOperationException(string.Format
                 ("Couldn't find view '{0}'", viewName));
            using var output = new StringWriter();
            var viewContext = new ViewContext(actionContext,
                view,
                new ViewDataDictionary<TModel>(metadataProvider:
                 new
                EmptyModelMetadataProvider(), modelState: new
                ModelStateDictionary())
            {
                Model = model
            },
new TempDataDictionary(actionContext.HttpContext,
 _tempDataProvider), output, new HtmlHelperOptions());
            await view.RenderAsync(viewContext);
            return output.ToString();
        }
```

After creating the `EmailViewEngine` class, extract its interface, `IEmailViewEngine`, as follows:

```
public interface IEmailViewEngine
    {
        Task<string> RenderEmailToString<TModel>(string
          viewName, TModel model);
    }
```


WOW! eBook
www.wowebook.org

2. Create a new folder called `Helpers` and add a new class to it called `EmailViewRenderHelper.cs`:

```
public class EmailViewRenderHelper
    {
        IWebHostEnvironment _hostingEnvironment;
        IConfiguration _configurationRoot;
        IHttpContextAccessor _httpContextAccessor;
        public async Task<string> RenderTemplate<T>(string
         template, IWebHostEnvironment hostingEnvironment,
        IConfiguration      configurationRoot,
         IHttpContextAccessor httpContextAccessor, T model
        ) where T : class
        {
            _hostingEnvironment = hostingEnvironment;
            _configurationRoot = configurationRoot;
            _httpContextAccessor = httpContextAccessor;
            var renderer = httpContextAccessor.HttpContext.
            RequestServices
            .GetRequiredService<IEmailViewEngine>();
            return await renderer.RenderEmailToString<T>(template,
             model);
        }
    }
```

3. Add a new service called `EmailTemplateRenderService` in the `Services` folder. It will have the following constructor:

```
public class EmailTemplateRenderService
    {
        private IWebHostEnvironment _hostingEnvironment;
        private IConfiguration _configuration;
        private IHttpContextAccessor _httpContextAccessor;
        public EmailTemplateRenderService(IWebHostEnvironment
         hostingEnvironment, IConfiguration configuration,
        IHttpContextAccessor httpContextAccessor)
        {
            _hostingEnvironment = hostingEnvironment;
            _configuration = configuration;
            _httpContextAccessor = httpContextAccessor;
        }
    }
```

Now, create a `RenderTemplate` method, as follows:

```
public async Task<string> RenderTemplate<T>(string templateName, T
model, string host) where T : class
        {
```


WOW! eBook
www.wowebook.org

```
            var html = await new EmailViewRenderHelper().
            RenderTemplate(templateName, _hostingEnvironment,
            _configuration, _httpContextAccessor, model);
            var targetDir = Path.Combine(Directory.
             GetCurrentDirectory(), "wwwroot", "Emails");
            if (!Directory.Exists(targetDir))
             Directory.CreateDirectory(targetDir);
            string dateTime = DateTime.Now.
             ToString("ddMMHHyyHHmmss");
            var targetFileName = Path.Combine(targetDir,
            templateName.Replace("/", "_").Replace("\\", "_")
              + "." + dateTime + ".html");
            html = html.Replace("{ViewOnLine}", $"
             {host.TrimEnd('/')}/Emails/{Path.GetFileName
             (targetFileName)}");
            html = html.Replace("{ServerUrl}", host);
            File.WriteAllText(targetFileName, html);
            return html;
        }
```

Extract its interface and name it `IEmailTemplateRenderService`.

4. Register `EmailViewEngine` and `EmailTemplateRenderService` in the `Startup` class:

```
services.AddTransient<IEmailTemplateRenderService,
 EmailTemplateRenderService>();
            services.AddTransient<IEmailViewEngine,
             EmailViewEngine>
             ();
```

Note that you need to register `EmailViewEngine` and `EmailTemplateRenderService` as transient because of `HTTPContextAccessor` injection.

5. Add a new layout page in the `Views/Shared` folder called `_LayoutEmail.cshtml`. First, we'll create the `head` section, as follows:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
     scale=1.0" />
    <title>@ViewData["Title"] - TicTacToe</title>
    <environment include="Development">
```


WOW! eBook
www.wowebook.org

```
        <link rel="stylesheet"
              href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
              href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7
              /css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css
               /bootstrap.min.css"
              asp-fallback-test-class="sr-only"
              asp-fallback-test-property="position"
              asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css"
          asp-append-version="true" />
    </environment>
</head>
```

Now, we'll create the `body` section, as follows:

```
<body>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer> <p>&copy; 2019 - TicTacToe</p> </footer>
    </div>
    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js">
        </script>
        <script src="~/js/site.js" asp-append-
version="true"></script>
    </environment>
    @RenderSection("Scripts", required: false)
</body>
```

6. Add a new model called `UserRegistrationEmailModel` to the `Models` folder:

```
public class UserRegistrationEmailModel
    {
        public string Email { get; set; }
        public string DisplayName { get; set; }
        public string ActionUrl { get; set; }
    }
```


WOW! eBook
www.wowebook.org

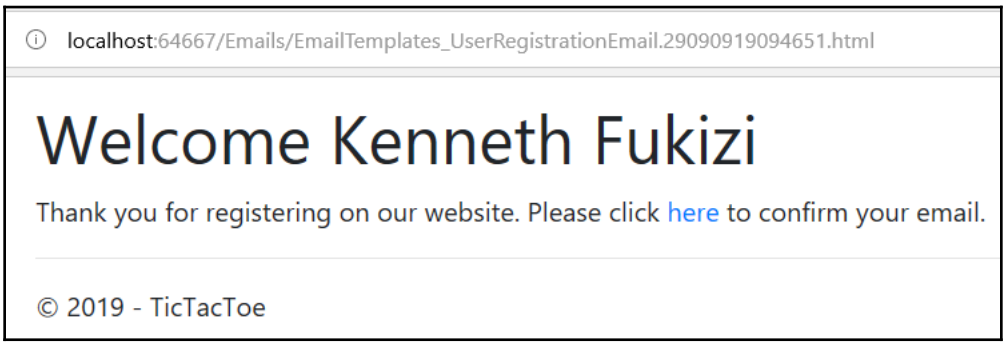
7. Create a new subfolder called `EmailTemplates` in the `Views` folder and add a new view called `UserRegistrationEmail`:

```
@model TicTacToe.Models.UserRegistrationEmailModel
@{
    ViewData["Title"] = "View";
    Layout = "_LayoutEmail";
}
<h1>Welcome @Model.DisplayName</h1>
Thank you for registering on our website. Please click <a
href="@Model.ActionUrl">here</a> to confirm your email.
```

8. Update the `EmailConfirmation` method within `UserRegistrationController` so that we can use the new email view engine before sending any emails:

```
var userRegistrationEmail = new UserRegistrationEmailModel
            {
                DisplayName = $"{user.FirstName} {user.LastName}",
                Email = email,
                ActionUrl = Url.Action(urlAction)
            };

            var emailRenderService = HttpContext.RequestServices.
              GetService<IEmailTemplateRenderService>();
            var message = await emailRenderService.RenderTemplate
             ("EmailTemplates/UserRegistrationEmail",
             userRegistrationEmail, Request.Host.ToString());
```

9. Start the application and register a new user. Open `UserRegistrationEmail` and analyze its content (look in the `wwwroot/Emails` folder):



WOW! eBook
www.wowebook.org

If you see the `InvalidOperationException: Unable to resolve service for type 'Microsoft.AspNetCore.Http.IHttpContextAccessor` error, you will need to register `IHttpContextAccessor` manually in the `Startup` class by adding `services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();` in the `ConfigureServices` method or by adding the built-in `services.AddHttpContextAccessor();` method.

You have looked at a variety of concepts and code examples throughout this book, but we still haven't talked about how to ensure excellent quality and maintainability for our applications. The next section is going to shed some light on this subject, which is dedicated to application testing.

Providing better quality by creating unit tests and integration tests

Building high-quality applications and satisfying application users is a difficult endeavor. Shipping products that have technical and functional flaws can lead to enormous problems during the maintenance phase of your applications.

The worst-case scenario is that, since maintenance is so demanding on time and resources, you won't be able to evolve your applications as quickly as possible to lower your time-to-market, and you will be unable to provide exciting new features. Don't think that your competition isn't waiting! They will surpass you and you will lose market shares and market leadership.

But how can you succeed? How can you reduce the time to detect bugs and functional problems? You have to test your code and your applications – and you have to do that as much as possible and as soon as possible. It is common knowledge that fixing a bug during development is cheaper and quicker, whereas fixing a bug during production takes more time and money.

Having a low **MTTR** (**short for mean time to repair**) for bugs can make a big difference when it comes to becoming a future market leader within your specific markets.

Let's divert a little and do some best practices housekeeping, which we will need to use in our application. In C#, we can check whether a string is null or empty using the `String.IsNullOrEmpty()` method. We need to do this because having a string that's null and having a string that's empty are two different scenarios altogether. An empty string is not necessarily null.

WOW! eBook
www.wowebook.org

There are also situations when we're dealing with collections and we need to check whether the collection is null or empty. Unfortunately, we don't have an out-of-the-box implementation like the one we used for strings, which means we'll create it ourselves.

Let's go to the extensions folder and create a static class called `CollectionsEtensionMethods` that contains two methods, as follows:

```
public static class CollectionsExtensionMethods
   {
       public static bool IsNullOrEmpty<T>(this IEnumerable<T>
        genericEnumerable)
       {
           return (genericEnumerable == null) ||
            (!genericEnumerable.Any());
       }

       public static bool IsNullOrEmpty<T>(this ICollection<T>
        genericCollection)
       {
           if (genericCollection == null)
           {
               return true;
           }
           return genericCollection.Count < 1;
       }
   }
```

Now, we'll be able to implement `IsNullOrEmpty()` checks on any of our collections, that is, as long as we reference the `TicTacToe.Extensions` namespace from anywhere in our application. We will see this in action in the following code snippet, where we will be looking at game session turns and finding out whether they are null or empty.

Let's continue with the development of the Tic-Tac-Toe application and learn how to carefully test it in more detail:

1. Add a new method called `AddTurn` to `GameSessionService` and update the game session service interface:

```
public async Task<GameSessionModel> AddTurn(Guid id, string email,
int x, int y)
{
  var gameSession = _sessions.FirstOrDefault(session => session.Id
    == id);
  List<TurnModel> turns;
  if (!gameSession.Turns.IsNullOrEmpty())
      turns = new List<TurnModel>(gameSession.Turns);
  else   turns = new List<TurnModel>();
```


WOW! eBook
www.wowebook.org

```
    turns.Add(new TurnModel {User = await _UserService.GetUserByEmail
      (email), X = x, Y = y });
    if (gameSession.User1?.Email == email) gameSession.ActiveUser =
      gameSession.User2;
    else  gameSession.ActiveUser = gameSession.User1;
    gameSession.TurnFinished = true;
    _sessions = new ConcurrentBag<GameSessionModel>(_sessions.Where(u
      => u.Id != id))
        { gameSession };
    return gameSession;
  }
```

2. Add a new method called `SetPosition` to `GameSessionController`:

```
public async Task<IActionResult> SetPosition(Guid id,
 string email, int x, int y)
{
    var gameSession =
    await _gameSessionService.GetGameSession(id);
    await _gameSessionService.AddTurn(gameSession.Id,
     email, x, y);
    return View("Index", gameSession);
}
```

3. Add a new model called `InvitationEmailModel` to the `Models` folder:

```
public class InvitationEmailModel
    {
        public string DisplayName { get; set; }
        public UserModel InvitedBy { get; set; }
        public DateTime InvitedDate { get; set; }
        public string ConfirmationUrl { get; set; }
    }
```

4. Add a new view called `InvitationEmail` to the `Views/EmailTemplates` folder:

```
@model TicTacToe.Models.InvitationEmailModel
@{
    ViewData["Title"] = "View";
    Layout = "_LayoutEmail";
}
<h1>Welcome @Model.DisplayName</h1>
You have been invited by @($"{Model.InvitedBy.FirstName} {
Model.InvitedBy.LastName} ") to play the Tic-Tac-Toe game.
Please click <a href="@Model.ConfirmationUrl">here</a> to join the
game.
```


WOW! eBook
www.wowebook.org

5. Update the `Index` method in `GameInvitationController` to be able to use the invitation email template we mentioned previously:

```
[HttpPost]
        public async Task<IActionResult> Index( GameInvitationModel
          gameInvitationModel, [FromServices]IEmailService
          emailService)
        {
            var gameInvitationService = Request.HttpContext.
             RequestServices.GetService<IGameInvitationService>();
            if (ModelState.IsValid)
            {
                try
                {
                    var invitationModel = new InvitationEmailModel
                    {
                        DisplayName = $"{gameInvitationModel.
                        EmailTo}",
                        InvitedBy = await
                         _userService.GetUserByEmail
                        ( gameInvitationModel.InvitedBy),
                        ConfirmationUrl =
                        Url.Action("ConfirmGameInvitation",
                         "GameInvitation",
                         new { id = gameInvitationModel.Id },
                         Request.Scheme, Request.Host.ToString()),
                         InvitedDate = gameInvitationModel.
                         ConfirmationDate
                    };
                    var emailRenderService = HttpContext.
                    RequestServices.GetService
                    <IEmailTemplateRenderService>();
                    var message = await emailRenderService.
                    RenderTemplate<InvitationEmailModel>
                    ("EmailTemplates/InvitationEmail",
                    invitationModel, Request.Host.ToString());
                    await emailService.SendEmail(
                        gameInvitationModel.EmailTo,
                        _stringLocalizer

                      ["Invitation for playing a Tic-Tac-Toe
                      game"], message);
                }
                catch
                {
                }
                var invitation = gameInvitationService.Add
                (gameInvitationModel).Result;
```


WOW! eBook
www.wowebook.org

```
                return RedirectToAction
                ("GameInvitationConfirmation", new { id =
                 gameInvitationModel.Id });
            }
            return View(gameInvitationModel);
        }
```

6. Add a new method called `ConfirmGameInvitation` to `GameInvitationController`:

```
[HttpGet]
     public IActionResult ConfirmGameInvitation(Guid id,
      [FromServices]IGameInvitationService
      gameInvitationService)
     {
         var gameInvitation = gameInvitationService.
         Get(id).Result;
         gameInvitation.IsConfirmed = true;
         gameInvitation.ConfirmationDate = DateTime.Now;
         gameInvitationService.Update(gameInvitation);
         return RedirectToAction("Index", "GameSession", new
          { id
          = id });
     }
```

7. Start the application and verify that everything is working as expected, including the various emails and steps for starting a new game.

Now that we have implemented all this new code, how do we test it? How do we ensure that it is working as expected? We could start the application in debug mode and verify that all the variables have been set correctly and that the application flow is correct, but that would be very tedious and not very efficient.

What would be better than doing this? Using unit tests and integration tests. We will look at these tests in the upcoming sections.

WOW! eBook
www.wowebook.org

Adding unit tests

Unit tests allow you to individually verify the behavior of your various technical components and ensure that they are working as expected. They also help you quickly identify regressions and analyze the overall impact of new developments. Visual Studio 2019 includes powerful features for unit testing.

The Test Explorer helps you run unit tests as well as view and analyze test results. For that, you can either use the built-in Microsoft testing framework or additional frameworks such as NUnit or xUnit.

Furthermore, you can automatically execute unit tests after each build so that developers can react quickly if something isn't working as expected.

Refactoring code can be done without fearing regressions since unit tests ensure that everything is still working like it was previously. No more excuses for not having the best code quality possible!

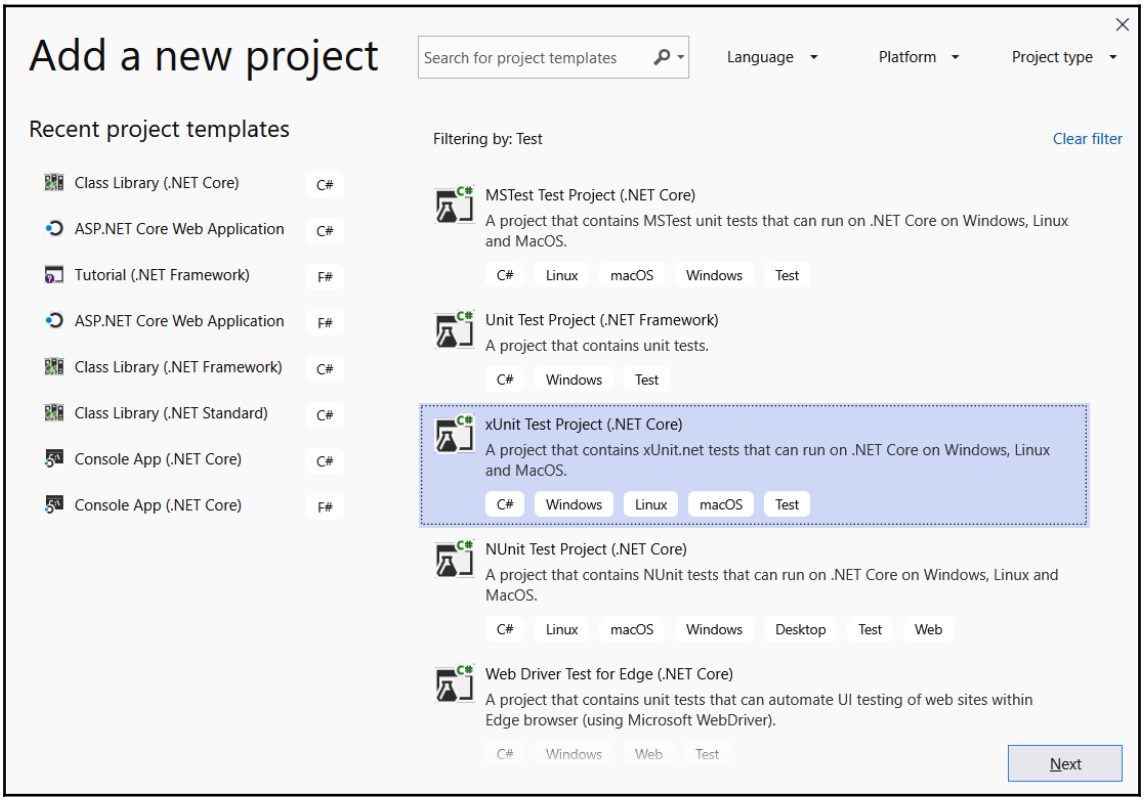
You could even go further and apply **test-driven development** (**TDD**), which is where you write unit tests before writing implementations. Additionally, unit tests become some sort of design document and functional specifications. A further step would be to apply **behavior-driven development** (**BDD**) and create tests from specifications.

This book is about ASP.NET Core 3, so we won't go into too much detail about unit tests. It is, however, advised to dig deeper and familiarize yourself with all the different unit test concepts so that you can build better applications.

WOW! eBook
www.wowebook.org

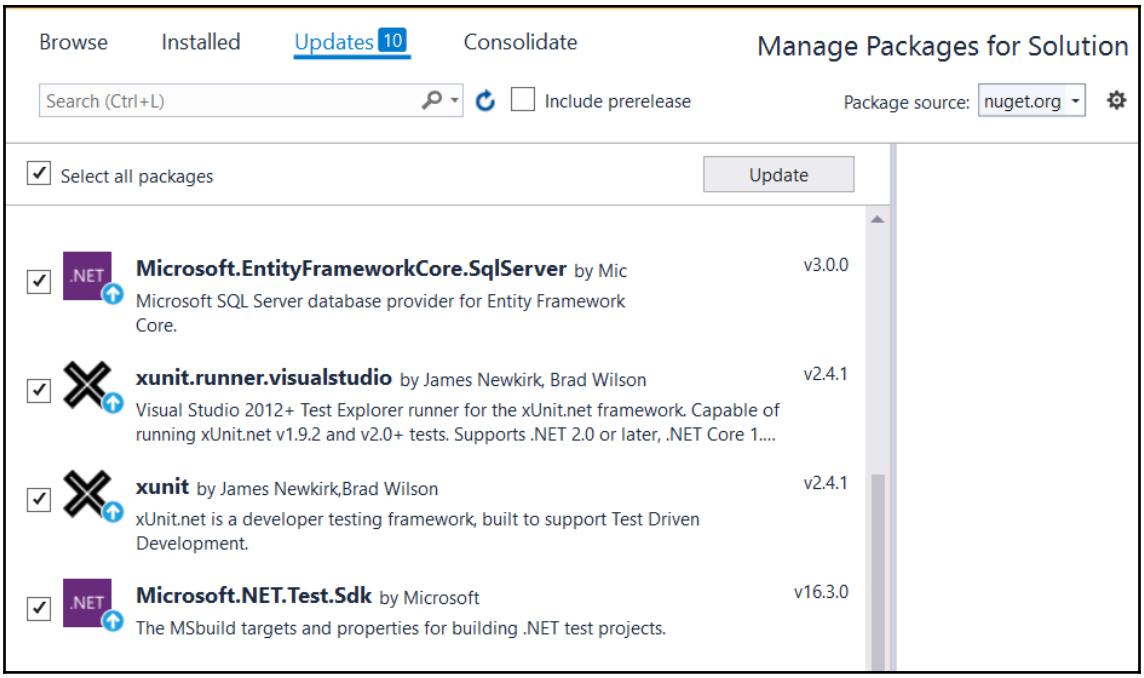
Let's learn how easy it is to use xUnit, which is the preferred unit testing framework for ASP.NET Core 3:

1. Add a new project of the **xUnit Test Project (.NET Core)** type called `TicTacToe.UnitTests` to the **TicTacToe** solution:

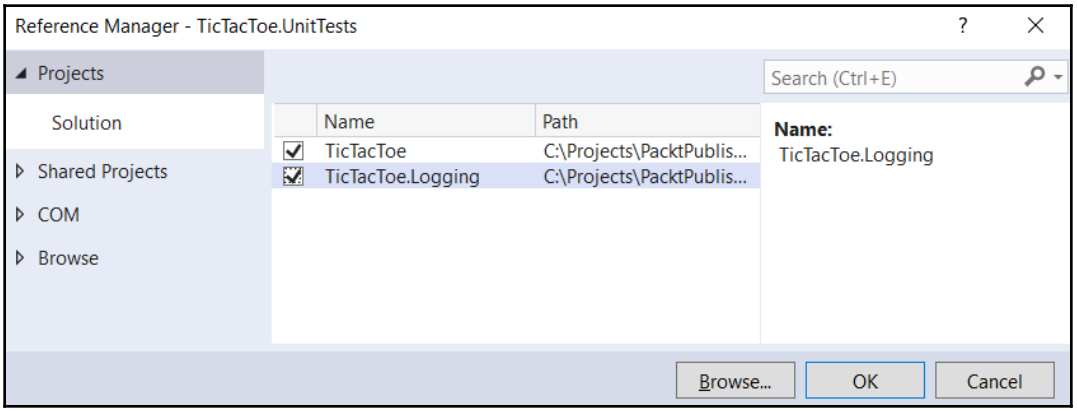


2. Update the **xunit** and **Microsoft.NET.Test.SDK** NuGet packages to their latest versions using the NuGet Package Manager:

WOW! eBook
www.wowebook.org



3. Add references to the **TicTacToe** and **TicTacToe.Logging** projects:



WOW! eBook
www.wowebook.org

4. Delete the autogenerated class, add a new class called `FileLoggerTests.cs` for testing a regular class, and implement a new method called `ShouldCreateALogFileAndAddEntry`:

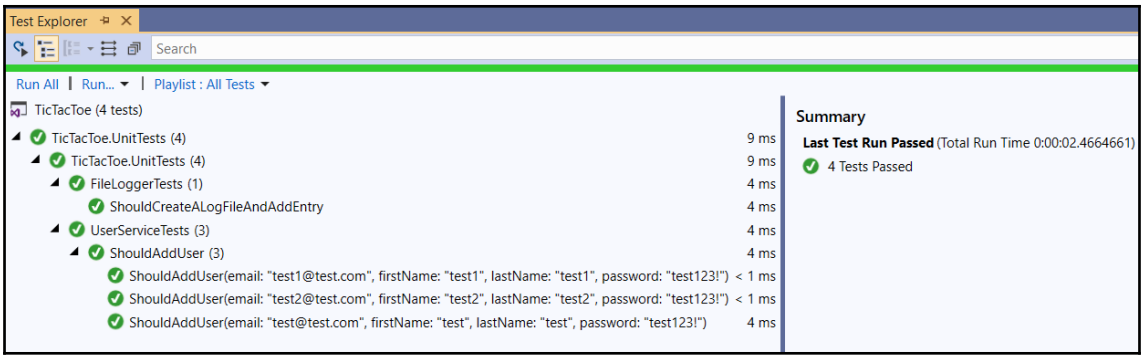
```
public class FileLoggerTests
    {
        [Fact]
        public void ShouldCreateALogFileAndAddEntry()
        {
            var fileLogger = new FileLogger(
            "Test", (category, level) => true,
            Path.Combine(Directory.GetCurrentDirectory(),
             "testlog.log"));
            var isEnabled = fileLogger.IsEnabled
             (LogLevel.Information);
            Assert.True(isEnabled);
        }
    }
```

5. Add another new class called `UserServiceTests.cs` for testing services and implement a new method called `ShouldAddUser`:

```
public class UserServiceTests
{
 [Theory]
 [InlineData("test@test.com", "test", "test", "test123!")]
 [InlineData("test1@test.com", "test1", "test1", "test123!")]
 [InlineData("test2@test.com", "test2", "test2", "test123!")]
 public async Task ShouldAddUser(string email, string firstName,
 string lastName, string password)
        { var userModel = new UserModel
            {   Email = email,
                FirstName = firstName,
                LastName = lastName,
                Password = password    };
            var userService = new UserService();
            var userAdded = await userService.RegisterUser
             (userModel);
            Assert.True(userAdded); }
    }
```

6. Open **Test Explorer** via **Test | Windows | Test Explorer** and choose to **Run All** to ensure that all the tests execute successfully:

WOW! eBook
www.wowebook.org



Unit tests are great and really important, but also somewhat limited. They only test each technical component separately, which is the main goal of this type of test.

The idea behind unit tests is to quickly get a glimpse of the current status of all your technical components, one by one, without slowing down the continuous integration process. They don't test applications under real production conditions since external dependencies are mocked. Instead, they are intended to run quickly and ensure that each method being tested creates no unintended side effects in other methods or classes. If you stop here, you won't be able to find as many bugs as you usually would during the development phase. You have to go even further and test all the components together in a real environment; this is where integration tests come into play.

Adding integration tests

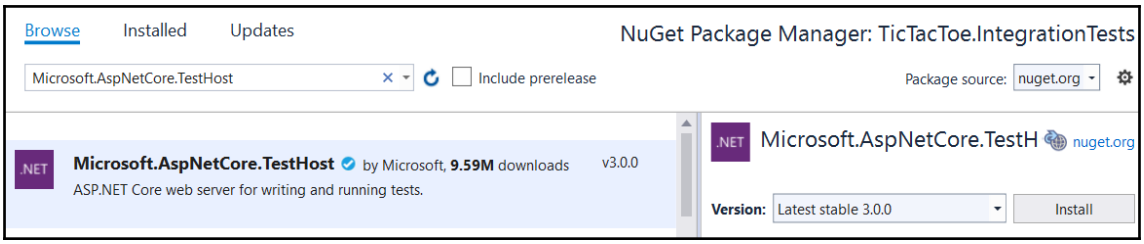
Integration tests are a logical extension of unit tests. They test the integration between multiple technical components within your applications in a real environment with access to external data sources (such as databases, web services, and caches). The goal of this type of test is to ensure that everything is working well together and providing the expected functionalities when combining various technical components to create application behavior.

Furthermore, integration tests should always have cleanup steps so that they can run repeatedly without error and don't leave any artifacts behind in databases or filesystems.

WOW! eBook
www.wowebook.org

In the following example, you will learn how to apply integration tests to the Tic-Tac-Toe demo application:

1. Add a new project of the **xUnit Test Project (.NET Core)** type called `TicTacToe.IntegrationTests` to the **TicTacToe Solution**, update the NuGet packages, and add references to the **TicTacToe** and **TicTacToe.Logging** projects as shown in the preceding unit tests project.
2. Add the `Microsoft.AspNetCore.TestHost` NuGet package to the `IntegrationTests` project, as shown in the following screenshot. This allows us to create fully automated integration tests using xUnit:



3. Delete the autogenerated class, add a new class called `IntegrationTests.cs`, and implement a new method called `ShouldGetHomePageAsync`:

```
[Fact]
public async Task ShouldGetHomePageAsync()
{
    var response = await _httpClient.GetAsync("/");
    response.EnsureSuccessStatusCode();
    var responseString = await response.Content.
        ReadAsStringAsync();
    Assert.Contains("Welcome to the Tic-Tac-Toe Desktop
     Game!", responseString);
}
```

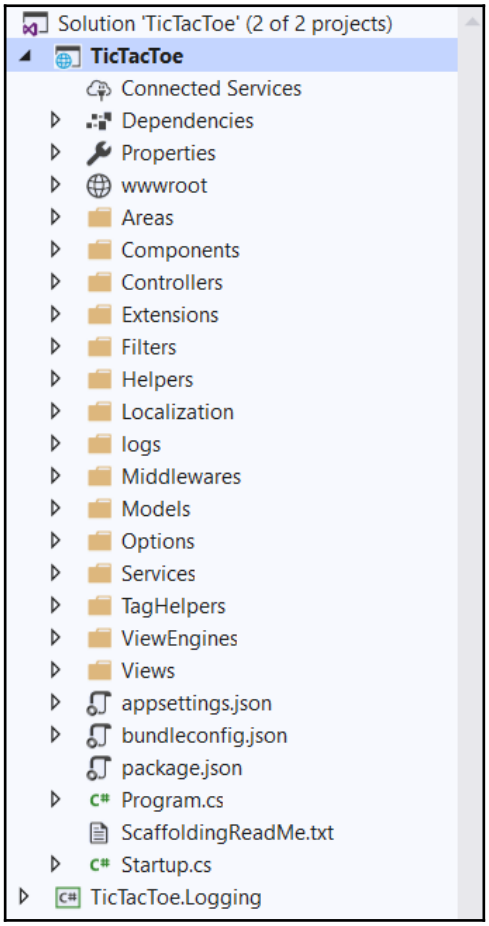
4. Run the tests in **Test Explorer** and ensure that they execute successfully.

Now that you have learned how to test your applications, you can continue to add additional unit and integration tests to fully understand these concepts and to build test coverage that will allow you to provide high-quality applications.

WOW! eBook
www.wowebook.org

Layering ASP.NET Core 3 applications

Some of you may have noticed that it may not always be a good idea to cram a lot of functionality into a single project. Our project's structure currently looks like this:



We have been adding folders upon folders and inside folders, and for large projects, it can quickly get out of hand and be a nightmare in terms of maintenance. This section serves only to give you awareness of the best practices to consider while designing our solutions using a layered architecture, which aims at achieving the following:

- A defined SoC.
- Less painful maintenance because of low coupling between layers and high cohesion between the layers.

WOW! eBook
www.wowebook.org

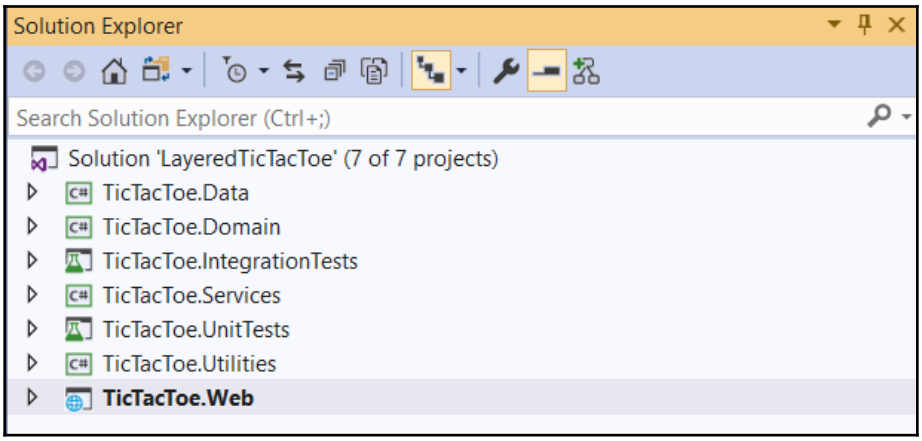
- The ability to be able to exchange and switch out different implementations of layer interfaces.
- Other solutions should be able to reuse functionality that's been exposed by the various layers.

Determining the required layers

For most of the applications that you will write, they will have common functionality. It is advised to group this common functionality into different projects as layers.

In the case of our `TicTacToe` demo application, we would have all the views grouped into one project as the presentation layer, services such as the `UserService`, `EmailService`, and others all grouped into the service layer, all the models such as `UserModel` grouped into a project as the domain layer, and a data access layer that will contain the database contexts. This will be explained in `Chapter 9`, *Accessing Data Using Entity Framework Core 3*.

The hypothetical layered application for our demo application would look as follows:



It's nice to have a layered application, but don't go on a wanton spree of randomly creating layers. The key is to look for functionality that makes sense for it to be grouped together so that you can make your application more maintainable and more scalable.

We won't be using this layered architecture in this book. We are only mentioning it for the sake of awareness so that you understand that we can improve on the current design of the `TicTacToe` application in many ways.

WOW! eBook
www.wowebook.org

Deciding on the distribution for layers and components

Layers and components should be distributed across separate physical tiers, but only where it is necessary to do so. There are several reasons for implementing distributed deployment and that includes security policies, physical constraints, shared business logic, and scalability.

For the sake of simplifying this book's content, and for the needs of the `TicTacToe` application, we are only deploying the application as a single instance, not distributed (we will cover this in `Chapter 12`, *Hosting ASP.NET Core 3 Applications*). Therefore, suffice to say that there are other ways and means of hosting an application with multiple instances as a distributed application.

In web applications, you should deploy the business layer and presentation layer components on the same physical tier to maximize performance and ease operational management, unless security restrictions need a trust boundary between them.

Determining rules for interactions between layers

When it comes to a layering strategy, rules must be defined for how the layers will interact with each other. The main reasons for specifying interaction rules are to minimize dependencies and eliminating circular references. For example, if two layers each have a dependency on components in the other layer, then circular dependencies will be introduced.

Only implement top-down interaction. Higher-level layers can interact with the layers below them, but a lower level layer should never interact with the layers above.

Implement a rule that will help you avoid circular dependencies between layers. Events can be used to make components in higher layers aware of changes in lower layers without introducing dependencies.

WOW! eBook
www.wowebook.org

Identifying cross-cutting concerns

When you separate a project into layers, you will notice some functionality that is repeatedly done in every layer. For example, you will find that you have to do validation functionality in each and every layer. Another example is that you have to authenticate every time in different layers. The best option is to identify these kinds of functionality and group them into one project as cross-cutting concerns.

The name of a cross-cutting concerns project (layer) does not have to be named exactly like that. You may choose to call the project your own name as you see fit. Other candidate functionalities that you could put in the cross-cutting layer include how you manage the application's exceptions, how you cache frequently used objects, and a logger functionality.

The advantage of getting these cross-cutting functionalities into one layer is that you're promoting reuse and that it makes our application a bit more maintainable.

Avoid mixing the cross-cutting code with code in the components of each layer so that the layers and their components only make calls to the cross-cutting components when they must carry out an action such as logging, caching, or authentication.

Summary

In this chapter, you learned about the MVC pattern, its different components and layers, and how important it is for building great ASP.NET Core 3 web applications.

You learned how to use layout pages and the features surrounding it to create device-specific layouts and thus adapt your user interfaces to the devices they will be running on. Furthermore, we used view pages to build the visible part, the presentation layer, of our web applications after learning about the different types of state management in ASP.NET Core 3.

Then, we discussed partial views, View Components, and Tag Helpers so that we can encapsulate and reuse our presentation logic throughout the different views of our applications. Toward the end, we illustrated advanced concepts such as the view engine, as well as unit tests and integration tests for creating high-quality applications with a low MTTR for bugs.

WOW! eBook
www.wowebook.org

Finally, we learned how important it is to structure our more complex applications using a layered architecture and the basics that we need to consider.

By reading this chapter, you can now create views, models, and controllers; detect mobile devices; use View Components; divide an application into areas; and decide what layers to create for your application.

In the next chapter, we will talk about the ASP.NET Core 3 web API framework and how to build, test, and deploy web API applications.

WOW! eBook
www.wowebook.org

8
Creating Web API Applications

You may not know it yet, but this chapter is the chapter you have been waiting for! It is very special for multiple reasons.

First, we will finish the gaming part and you will be able to start playing the Tic-Tac-Toe game. Yes – at last, the whole application will be up and running and you will be able to compete against other users. Very exciting!

Secondly, you will learn how to integrate your applications with other systems and services. This is very important since modern applications are no longer isolated silos. Instead, they communicate with each other and continuously exchange data to provide even more value to customers. How can we do this? We can provide interoperable **web application programming interfaces** (**web APIs**), which allow users to plug in components, sometimes based on completely different technologies!

Thirdly, using web APIs will not only allow you to integrate with other systems; it will also help you build more flexible and reusable application components, which you can then combine to create new applications that respond to more advanced use cases.

The APIs we will be creating in this chapter are not only usable by the MVC web frontend we have been working on, but also by any new mobile frontends you may build in the future. This will allow you to reach even more customers. You will be able to provide omnichannel experiences to your customers, where they start using one device and finish on another.

WOW! eBook
www.wowebook.org

In this chapter, we will cover the following topics:

- Applying web API concepts and best practices
- Building RPC, REST, and HATEOAS-style web APIs
- Web API security
- ASP.NET Core web API help pages with Swagger/OpenAPI

Technical requirements

The source code for this chapter can be found at `https://github.com/PacktPublishing/Learn-ASP.NET-Core-3-Second-Edition/tree/master/Chapter08`.

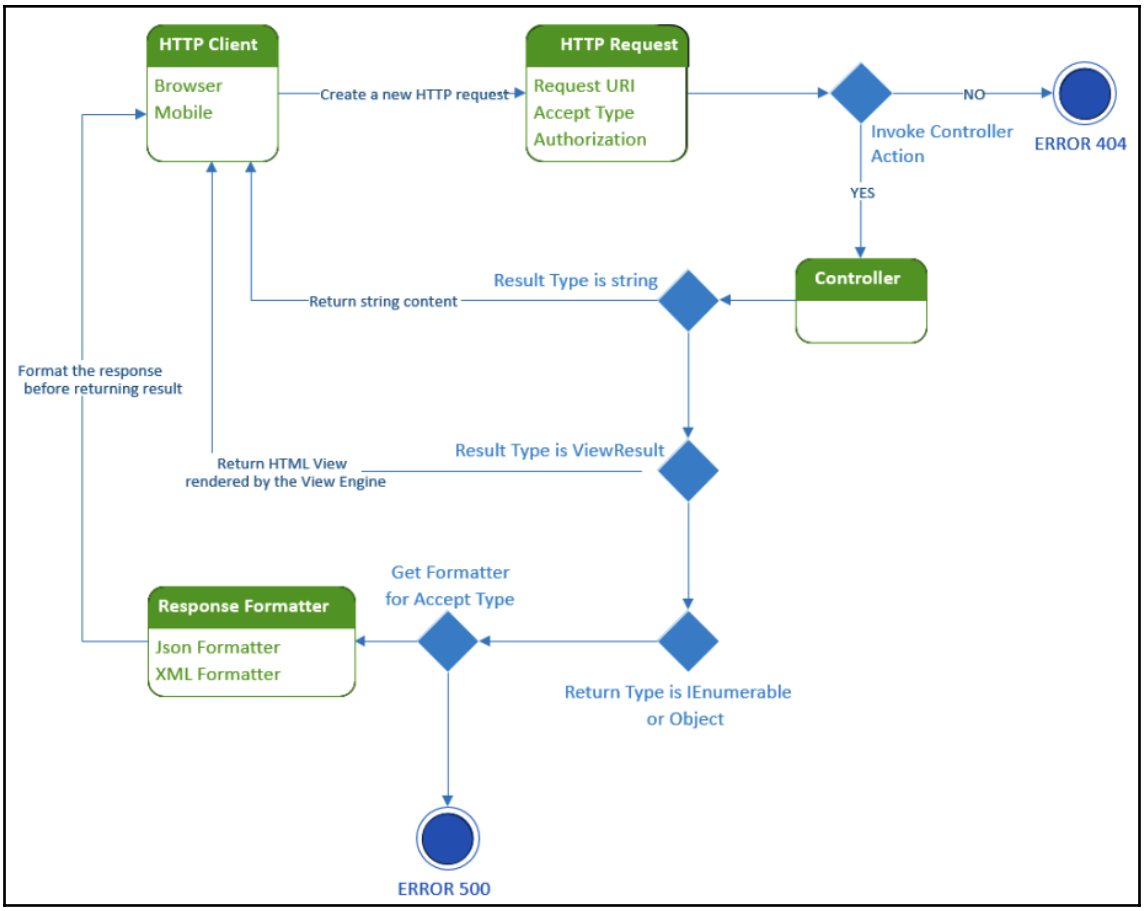
Applying web API concepts and best practices

ASP.NET Core 3 combines the best features of ASP.NET MVC and web APIs into a single framework. This makes complete sense since they provide many similar functionalities.

Before this merger, developers had to rewrite code when they needed to expose data in different formats via MVC and web APIs. They had to work with multiple frameworks and concepts at the same time. Fortunately, this entire process has been completely streamlined in ASP.NET Core 3, as you will see in this chapter.

The following diagram illustrates how client HTTP requests are handled by ASP.NET Core 3 in terms of web APIs and MVC:

WOW! eBook
www.wowebook.org



Web APIs normally use either JSON or XML as a response format. JSON is the preferred format since it has become a quasi-standard on the market and most modern applications use it due to its simplicity and efficiency.

Furthermore, filters and middleware can be used with web APIs since ASP.NET Core 3 manages web APIs the same way it does for standard MVC Controllers. This can be quite handy in some use cases and developers can apply their skills more widely.

In general, there are three different styles for creating web APIs when using ASP.NET Core 3:

- RPC-style
- REST-style
- HATEOAS-style

WOW! eBook
www.wowebook.org

Note that it is also possible to use the **Simple Object Access Protocol** (**SOAP**) to create web APIs, but it is not recommended. Instead, SOAP should be used in the context of standard web services, which is why it is not shown in the following examples.

We will present each style in more detail, along with some practical examples, which will help you decide on your own integration strategy.

Building RPC-style web APIs

The **RPC**-style is based on the **Remote Procedure Call** paradigms, which have existed for a long time now (since the early 1980s). It is based on including an action name in the URL, which makes it very similar to standard MVC actions.

One of the big advantages of ASP.NET Core 3 is that you do not need to separate the MVC parts from the web API parts. Instead, you can use both in your controller implementations.

Controllers are now capable of rendering view results, as well as JSON/XML API responses, which enables easy migrations from one to the other. Additionally, you can use a specific route path or the same route path for your MVC actions.

In the following example, you are going to transform a controller action from an MVC view result into an RPC-style web API:

1. Add a new method called `ConfirmEmail` to `UserRegistrationController`; it will be used to confirm the user registration email. The method accepts an email as a parameter, gets the user by the supplied email, and if the user is found, it updates the fact that the user has had their email confirmed and sets the timestamp of when it was confirmed:

```
[HttpGet]
public async Task<IActionResult> ConfirmEmail(string email)
{
  var user = await _userService.GetUserByEmail(email);
  if (user != null)
  {
    user.IsEmailConfirmed = true;
    user.EmailConfirmationDate = DateTime.Now;
    await _userService.UpdateUser(user);
    return RedirectToAction("Index", "Home");
  }
  return BadRequest();
}
```


WOW! eBook
www.wowebook.org

2. Update the `ConfirmGameInvitation` method within `GameInvitationController`, store the email of the invited user in a session variable, and register the new user via the user service:

```
        [HttpGet]
        public async Task<IActionResult> ConfirmGameInvitation
         (Guid id,
         [FromServices]IGameInvitationService
          gameInvitationService)
        {
          var gameInvitation = await gameInvitationService.Get(id);
          gameInvitation.IsConfirmed = true;
          gameInvitation.ConfirmationDate = DateTime.Now;
          await gameInvitationService.Update(gameInvitation);
          Request.HttpContext.Session.SetString("email",
            gameInvitation.EmailTo);
          await _userService.RegisterUser(new UserModel
          {
            Email = gameInvitation.EmailTo, EmailConfirmationDate =
             DateTime.Now, IsEmailConfirmed =true
          });
          return RedirectToAction("Index", "GameSession", new { id
});
        }
```

3. Update the table element in `GameSessionViewComponent`, which can be found inside the `Views/Shared/Components/GameSession/default.cshtml` file, by removing the `@if (Model.ActiveUser?.Email == email)` wrap. Next, instead of wrapping the table element with a `gameBoard` div element (as shown in the following code), update the wait turn `div` element, which has an `id` called `"divAlertWaitTurn"`, as follows:

```
 <div id="gameBoard">
    <table>
        ...
    </table>
</div>
<div class="alert" id="divAlertWaitTurn">
    <i class="glyphicon glyphicon-alert">Please wait until
     the other user has finished his turn.</i>
</div>
```


WOW! eBook
www.wowebook.org

4. Add a new JavaScript file within the `wwwroot\app\js` folder called `GameSession.js`. This will be used to call the web API. The `SetGameSession` method accepts a session ID, which is used for the setting the game session:

```
function SetGameSession(gdSessionId, strEmail) {
          window.GameSessionId = gdSessionId;
          window.EmailPlayer = strEmail;
        }

        $(document).ready(function () {
          $(".btn-SetPosition").click(function () {
            var intX = $(this).attr("data-X");
            var intY = $(this).attr("data-Y");
            SendPosition(window.GameSessionId, window.EmailPlayer,
             intX, intY);
          })
        })
```

Then, send the position, as follows:

```
function SendPosition(gdSession, strEmail, intX, intY) {
          var port = document.location.port ? (":" +
           document.location.port) : "";
          var url = document.location.protocol + "//" +
           document.location.hostname + port +
           "/restApi/v1/SetGamePosition/" + gdSession;
          var obj = {
            "Email": strEmail, "x": intX, "y": intY
          };
```

Add a temporary alert box for testing purposes:

```
          var json = JSON.stringify(obj);
          $.ajax({
            'url': url,
            'accepts': "application/json; charset=utf-8",
            'contentType': "application/json",
            'data': json,
            'dataType': "json",
            'type': "POST",
            'success': function (data) {
              alert(data);
            }
          });
        }
```


WOW! eBook
www.wowebook.org

5. Add the preceding JavaScript file to the `bundleconfig.json` file so that you can bundle it with the other files into the `site.js` file:

```
{
  "outputFileName": "wwwroot/js/site.js",
  "inputFiles": [
    "wwwroot/app/js/scripts1.js",
    "wwwroot/app/js/scripts2.js",
    "wwwroot/app/js/GameSession.js"
  ],
  "sourceMap": true,
  "includeInProject": true
},
```

6. Add a new property called `Email` to the `TurnModel` model:

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

7. Update the `SetPosition` method within `GameSessionController`. Here, expose it as a web API so that you can receive AJAX calls from the JavaScript `SendPosition` function we implemented previously:

```
[Produces("application/json")]
[HttpPost("/restapi/v1/SetGamePosition/{sessionId}")]
public async Task<IActionResult> SetPosition([FromRoute]Guid
sessionId)
{
    if (sessionId != Guid.Empty)
    {
        using (var reader = new StreamReader(Request.Body,
         Encoding.UTF8, true, 1024, true))
        {
            ...
        }
    }
    return BadRequest("Id is empty");
}
```

Then, add the following code to the `StreamReader` body:

```
var bodyString = reader.ReadToEnd();
if (string.IsNullOrEmpty(bodyString))
  return BadRequest("Body is empty");
var turn = JsonConvert.DeserializeObject<TurnModel>(bodyString);
   turn.User = await HttpContext.RequestServices.
   xGetService<IUserService>().GetUserByEmail(turn.Email);
   turn.UserId = turn.User.Id;
```


WOW! eBook
www.wowebook.org

```
if (turn == null) return BadRequest("You must pass a TurnModel
    object in your body");
var gameSession = await _gameSessionService.
    GetGameSession(sessionId);
if (gameSession == null)
  return BadRequest($"Cannot find Game Session {sessionId}");
if (gameSession.ActiveUser.Email != turn.User.Email)
    return BadRequest($"{turn.User.Email} cannot play this turn");
    gameSession = await _gameSessionService.
    AddTurn(gameSession.Id, turn.User.Email, turn.X, turn.Y);
if (gameSession != null && gameSession.ActiveUser.Email !=
    turn.User.Email)
   return Ok(gameSession);
else
   return BadRequest("Cannot save turn");
```

Note that it is good practice to prefix web APIs with a meaningful name and a version number (for example, `/restapi/v1`), as well as support for JSON and XML.

8. Update the **Game Session Index View** in the `Views` folder and call the JavaScript `SetGameSession` function with the corresponding parameters:

```
@using Microsoft.AspNetCore.Http
@model TicTacToe.Models.GameSessionModel
@{
  var email = Context.Session.GetString("email");
}
@section Desktop {
  ...
}
@section Mobile{
  ...
}
<h3>User Email @email</h3>
<h3>Active User <span id="activeUser">
 @Model.ActiveUser?.Email</span></h3>
<vc:game-session game-session-id="@Model.Id"></vc:game-
session>
@section Scripts{
  <script>  SetGameSession("@Model.Id", "@email");
  </script>
}
```


WOW! eBook
www.wowebook.org

9. Update the `ProcessEmailConfirmation` method for WebSockets in the communication middleware:

```
public async Task ProcessEmailConfirmation(HttpContext
 context,
 WebSocket currentSocket, CancellationToken ct, string
  email)
{
  var user = await _userService.GetUserByEmail(email);
  while (!ct.IsCancellationRequested &&
   !currentSocket.CloseStatus.HasValue &&
    user?.IsEmailConfirmed == false)
  {
    await SendStringAsync(currentSocket,
    "WaitEmailConfirmation", ct);
    await Task.Delay(500);
    user = await _userService.GetUserByEmail(email);
  }

  if (user.IsEmailConfirmed)
    await SendStringAsync(currentSocket, "OK", ct);
}
```

10. Update the `ProcessGameInvitationConfirmation` method for WebSockets in the communication middleware:

```
public async Task ProcessEmailConfirmation(HttpContext
context, WebSocket currentSocket, CancellationToken ct,
 string email)
{
    var user = await _userService.GetUserByEmail(email);
    while (!ct.IsCancellationRequested &&
     !currentSocket.CloseStatus.HasValue && user?
     .IsEmailConfirmed == false)
    {
        await SendStringAsync(currentSocket,
         "WaitEmailConfirmation", ct);
        await Task.Delay(500);
        user = await _userService.GetUserByEmail(email);
    }

    if (user.IsEmailConfirmed)
        await SendStringAsync(currentSocket, "OK", ct);
}
```


WOW! eBook
www.wowebook.org

11. Update the `CheckGameInvitationConfirmationStatus` method in the `scripts2.js` JavaScript file. It has to verify the returned data:

```
function CheckGameInvitationConfirmationStatus(id) {
  $.get("/GameInvitationConfirmation?id=" + id, function
    (data) {
    if (data.result === "OK") {
      if (interval !== null) {
        clearInterval(interval);
      }
      window.location.href = "/GameSession/Index/" + id;
    }
  });
}
```

12. Update the `Process` method in the Gravatar Tag Helper and handle the case where no photo exists correctly:

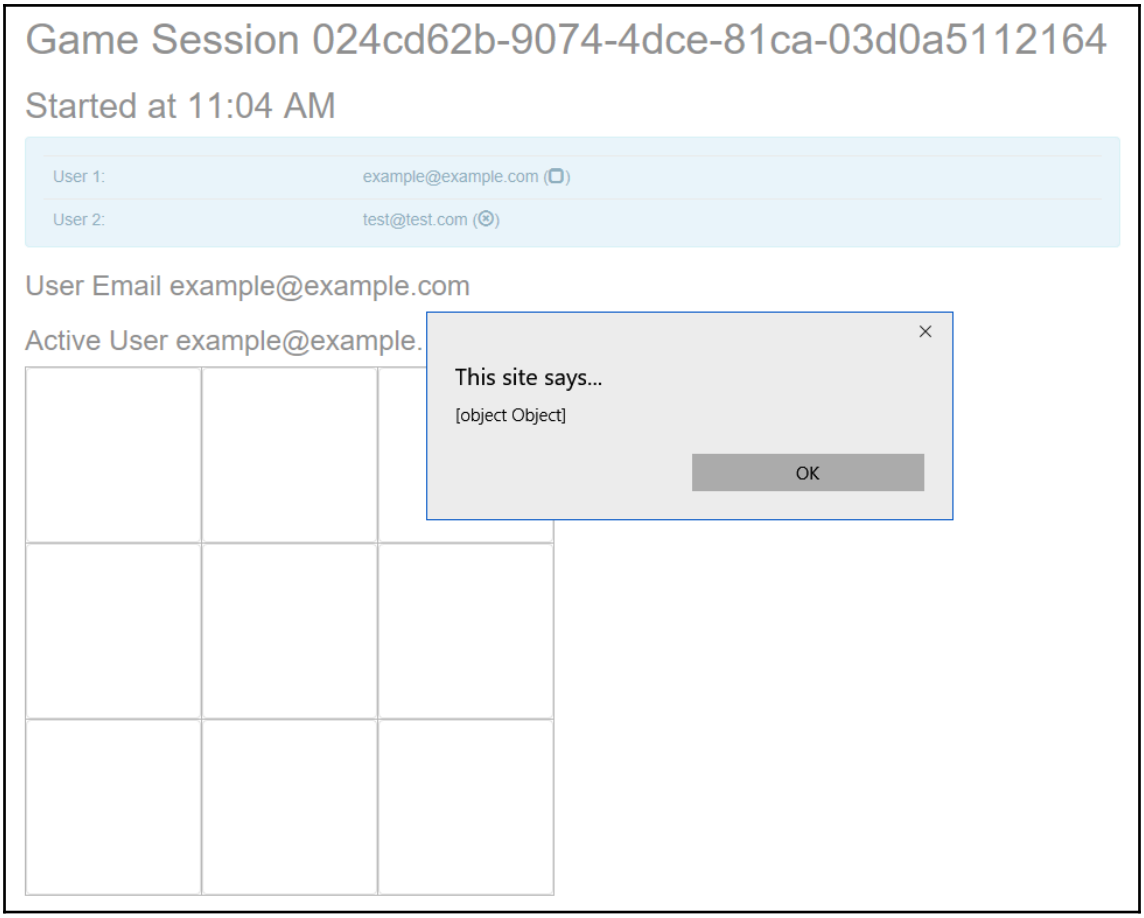
```
public override void Process(TagHelperContext context,
 TagHelperOutput output)
{
    byte[] photo = null;
    if (CheckIsConnected()) photo = GetPhoto(Email);
    else
    {
        string filePath = Path.Combine(Directory.
        GetCurrentDirectory(),"wwwroot", "images",
        "no-photo.jpg");
        if (File.Exists(filePath))  photo =
         File.ReadAllBytes(filePath);
    }
    if (photo != null && photo.Length > 0)
    {
        output.TagName = "img";
        output.Attributes.SetAttribute("src",
        $"
        {Convert.ToBase64String(photo)}");
    }
}
```

13. Update the `Add` method in `GameInvitationService`:

```
public Task<GameInvitationModel> Add(GameInvitation
Model gameInvitationModel)
{
  _gameInvitations.Add(gameInvitationModel);
  return Task.FromResult(gameInvitationModel);
}
```


WOW! eBook
www.wowebook.org

14. Update the **Desktop Layout Page** and **Mobile Layout Page**. Clean this up by removing the development `environment` tag containing `script1.js` and `script2.js` at the bottom of both pages.
15. Update the `scripts1.js` JavaScript file and clean up the previous unnecessary code by removing all the alert boxes that display whether WebSockets are enabled.
16. Start the application, register a new user, start a game session by inviting another user, and click on a cell. Now, you will see a JavaScript alert box:



WOW! eBook
www.wowebook.org

So far, you have learned how to transform the existing `GameSessionController` action into an RPC-style web API. Since all the different ASP.NET web frameworks have been centralized into a single framework in ASP.NET Core 3, this can be done easily and quickly without rewriting any code or changing your existing code too much.

In the next step, we will learn how to add a new method to the RPC-style web API to check if the turn for the current user has finished, which means that the next user can start their turn:

1. Add a new property called `TurnNumber` to `GameSessionModel` in order to track the current turn number:

```
public int TurnNumber { get; set; }
```

2. Add a new property called `IconNumber` to `TurnModel` so that you can define what icon (`X` or `O`) needs to be used for display later:

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

3. Add a new method called `GetGameSession`, which uses the game session service to get a game session, to the `GameSessionController`; it will be exclusive to web API calls:

```
[Produces("application/json")]
[HttpGet("/restapi/v1/GetGameSession/{sessionId}")]
public async Task<IActionResult> GetGameSession(Guid
 sessionId)
{
    if (sessionId != Guid.Empty)
    {
        var session = await _gameSessionService.
        GetGameSession(sessionId);

        if (session != null)
            return Ok(session);
        else
            return NotFound($"cannot found session
            {sessionId}");
    }
    else
        return BadRequest("session id is null");
}
```


WOW! eBook
www.wowebook.org

4. Update the `AddTurn` method in `GameSessionService` so that it calculates the `IconNumber` and `TurnNumber`. To do this, replace the following line of code:

```
turns.Add(new TurnModel {
           User = await _UserService.GetUserByEmail(email), X = x,
             Y = y });
```

Write the following code, which allows an icon number to be set:

```
public async Task<GameSessionModel> AddTurn(Guid id,
        string email, int x, int y)
        {
          ...
          turns.Add(new TurnModel
          {
            User = await _UserService.GetUserByEmail(email),
             X = x,
             Y = y,
             IconNumber = email == gameSession.User1?.
              Email ? "1" : "2"
          });

          gameSession.Turns = turns;
          gameSession.TurnNumber = gameSession.TurnNumber + 1;
          ...
        }
```

5. Update the **Game Session Index View**, user images, and add the possibility to enable and disable the gameboard by replacing the scripts section at the bottom with the following code snippet. This enables or disables the board, depending on whether a user is active or not:

```
@section Scripts{
  <script>
  SetGameSession("@Model.Id", "@email");
  EnableCheckTurnIsFinished();
  @if(email != Model.ActiveUser?.Email)
  {
    <text>DisableBoard(@Model.TurnNumber);</text>
  }
  else
  {
    <text>EnableBoard(@Model.TurnNumber);</text>
  }
  </script>
}
```


WOW! eBook
www.wowebook.org

6. Add a new JavaScript file called `CheckTurnIsFinished.js` to the `wwwroot\app\js` folder using the following `EnableCheckTurnIsFinished()` function. This checks whether a playing turn has finished:

```
function EnableCheckTurnIsFinished() {
          interval = setInterval(() => {CheckTurnIsFinished();},
            2000);
        }
        function CheckTurnIsFinished() {
          var port = document.location.port ? (":" +
            document.location.port) : "";
          var url = document.location.protocol + "//" +
            document.location.hostname + port +
           "/restapi/v1/GetGameSession/" + window.GameSessionId;

          $.get(url, function (data) {
            if (data.turnFinished === true && data.turnNumber >=
             window.TurnNumber) {
              CheckGameSessionIsFinished();
              ChangeTurn(data);
            }
          });
        }
```

In the same `CheckTurnIsFinished.js` file, add a `ChangeTurn()` function. This changes the turn of a player and disables or enables the board accordingly:

```
function ChangeTurn(data) {
          var turn = data.turns[data.turnNumber-1];
          DisplayImageTurn(turn);

          $("#activeUser").text(data.activeUser.email);
          if (data.activeUser.email !== window.EmailPlayer) {
            DisableBoard(data.turnNumber);
          }
          else {
            EnableBoard(data.turnNumber);
          }
        }
```


WOW! eBook
www.wowebook.org

Add the actual functionality to disable and enable the board, as follows:

```
function DisableBoard(turnNumber) {
   var divBoard = $("#gameBoard");
   divBoard.hide();
   $("#divAlertWaitTurn").show();
   window.TurnNumber = turnNumber;
 }

 function EnableBoard(turnNumber) {
   var divBoard = $("#gameBoard");
   divBoard.show();
   $("#divAlertWaitTurn").hide();
   window.TurnNumber = turnNumber;
 }
```

Finally, add a `DisplayImageTurn` function, which manipulates the cascading style sheets according to a respective turn, as follows:

```
function DisplayImageTurn(turn) {
          var c = $("#c_" + turn.y + "_" + turn.x);
          var css;

          if (turn.iconNumber === "1") {
          css = 'glyphicon glyphicon-unchecked';
        }
        else {
          css = 'glyphicon glyphicon-remove-circle';
        }

        c.html('<i class="' + css + '"></i>');
      }
```

Update `bundleconfig.json` so that it includes the new `CheckTurnIsFinished.js` file:

```
{
      "outputFileName": "wwwroot/js/site.js",
      "inputFiles": [
        "wwwroot/app/js/scripts1.js",
        "wwwroot/app/js/scripts2.js",
        "wwwroot/app/js/GameSession.js",
        "wwwroot/app/js/CheckTurnIsFinished.js"
      ],
      "sourceMap": true,
      "includeInProject": true
    },
```


WOW! eBook
www.wowebook.org

7. Update the `SetGameSession` method in the `GameSession.js` JavaScript file. Now, set `TurnNumber` to `0` by default:

```
function SetGameSession(gdSessionId, strEmail) {
  window.GameSessionId = gdSessionId;
  window.EmailPlayer = strEmail;
  window.TurnNumber = 0;
}
```

8. Update the `SendPosition` function in the `GameSession.js` JavaScript file and remove the temporary testing alert box we added previously. The game will be fully functional by the end of this section:

```
// Remove this alert
'success': function (data) {
            alert(data);
        }
```

9. Now, we need to add two new methods to `GameSessionController`. The first one is called `CheckGameSessionIsFinished` and uses the game session service to get the session and decide whether the game was a draw or was won by user `1` or `2`. As a result, the system will know whether the game session has finished. To do this, use the following code:

```
[Produces("application/json")]
[HttpGet("/restapi/v1/CheckGameSessionIsFinished/{sessionId}")]
public async Task<IActionResult> CheckGameSessionIsFinished(Guid
sessionId)
{ if (sessionId != Guid.Empty)
    {
      var session = await
        _gameSessionService.GetGameSession(sessionId);
      if (session != null)
      {
          if (session.Turns.Count() == 9) return Ok("The
           game was a draw.");
          var userTurns = session.Turns.Where(x => x.User ==
            session.User1).ToList();
          var user1Won = CheckIfUserHasWon(session.User1?.Email,
            userTurns);
          if (user1Won) return Ok($"{session.User1.Email} has
           won the game.");
          else
             {
               userTurns = session.Turns.Where(x => x.User ==
                session.User2).ToList();
               var user2Won = CheckIfUserHasWon(session.User2?.
```


WOW! eBook
www.wowebook.org

```
                    Email, userTurns);

                   if (user2Won)return Ok($"{session.User2.Email}
                    has won the game.");
                   else return Ok("");
                }
            }
            else
                return NotFound($"Cannot find session {sessionId}.");
            }
        else
            return BadRequest("SessionId is null.");
    }
```

Now, we need to implement the second method, that is, `CheckIfUserHasWon`, which determines whether a user has won the game and sends this information to `GameSessionController`:

```
private bool CheckIfUserHasWon(string email,
 List<TurnModel> userTurns)
        {
          if (userTurns.Any(x => x.X == 0 && x.Y == 0) &&
            userTurns.Any(x => x.X == 1 && x.Y == 0) &&
            userTurns.Any(x => x.X == 2 && x.Y == 0))
               return true;
          else if (userTurns.Any(x => x.X == 0 && x.Y == 1) &&
            userTurns.Any(x => x.X == 1 && x.Y == 1) &&
            userTurns.Any(x => x.X == 2 && x.Y == 1))
               return true;
          else if (userTurns.Any(x => x.X == 0 && x.Y == 2) &&
            userTurns.Any(x => x.X == 1 && x.Y == 2) &&
            userTurns.Any(x => x.X == 2 && x.Y == 2))
               return true;
          else if (userTurns.Any(x => x.X == 0 && x.Y == 0) &&
            userTurns.Any(x => x.X == 0 && x.Y == 1) &&
            userTurns.Any(x => x.X == 0 && x.Y == 2))
               return true;
          else if (userTurns.Any(x => x.X == 1 && x.Y == 0) &&
            userTurns.Any(x => x.X == 1 && x.Y == 1) &&
            userTurns.Any(x => x.X == 1 && x.Y == 2))
               return true;
          else if (userTurns.Any(x => x.X == 2 && x.Y == 0) &&
            userTurns.Any(x => x.X == 2 && x.Y == 1) &&
            userTurns.Any(x => x.X == 2 && x.Y == 2))
               return true;
          else if (userTurns.Any(x => x.X == 0 && x.Y == 0) &&
            userTurns.Any(x => x.X == 1 && x.Y == 1) &&
            userTurns.Any(x => x.X == 2 && x.Y == 2))
```


WOW! eBook
www.wowebook.org

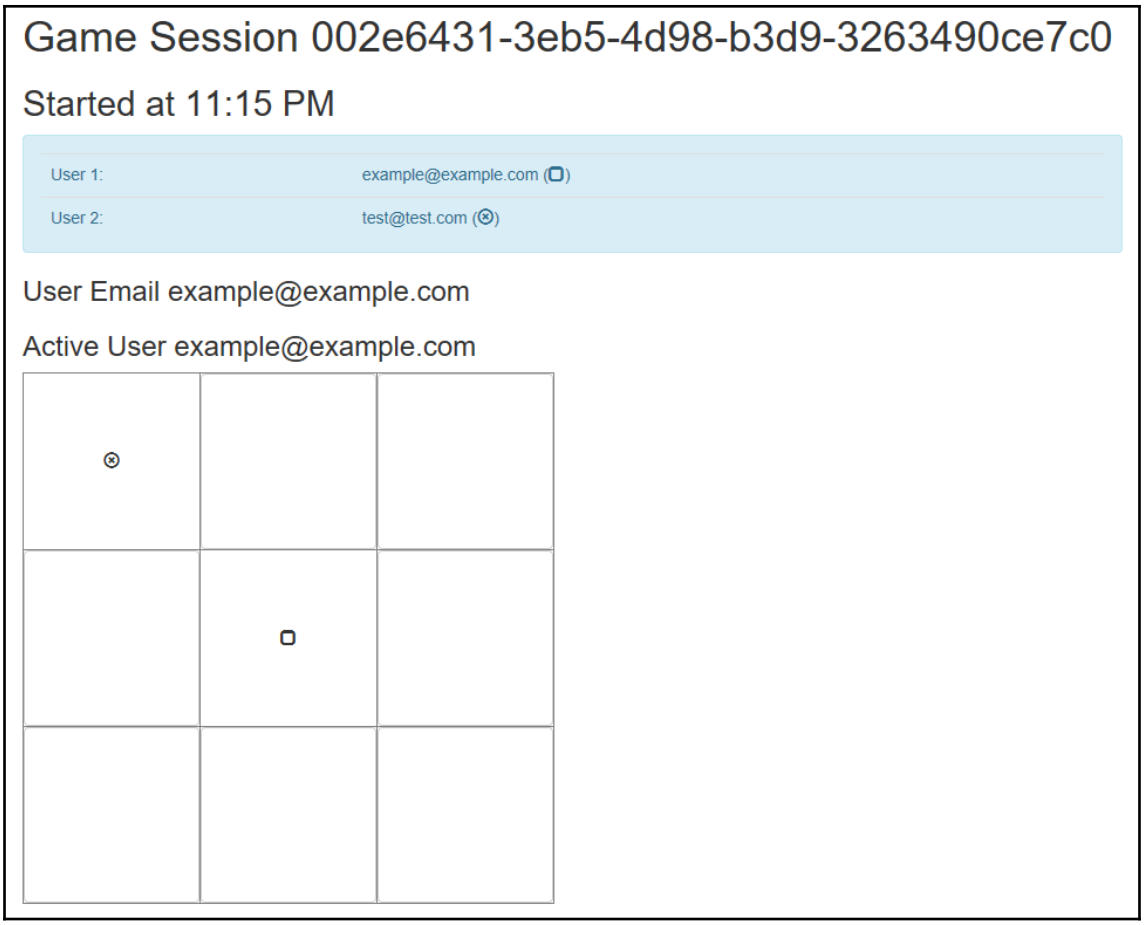
```
            return true;
        else if (userTurns.Any(x => x.X == 2 && x.Y == 0) &&
          userTurns.Any(x => x.X == 1 && x.Y == 1) &&
          userTurns.Any(x => x.X == 0 && x.Y == 2))
            return true;
        else
          return false;
      }
```

10. Add a new JavaScript file called `CheckGameSessionIsFinished.js` to the `wwwroot\app\js` folder and update the `bundleconfig.json` file accordingly:

```
function CheckGameSessionIsFinished() {
  var port = document.location.port ? (":" +
    document.location.port) : "";
  var url = document.location.protocol + "//" +
   document.location.hostname + port +
   "/restapi/v1/CheckGameSessionIsFinished/" +
     window.GameSessionId;

  $.get(url, function (data) {
    debugger;
    if (data.indexOf("won") > 0 || data == "The game
     was a draw.") {
      alert(data);
      window.location.href = document.location.protocol +
      "//" + document.location.hostname + port;
    }
  });
}
```

11. Start the game, register a new account, open the confirmation email, confirm it, send a game invitation email, confirm the game invitation, and start playing. Everything should be working now, and you should be able to play the game until a user has won or until the game ends in a draw:

WOW! eBook
www.wowebook.org



In this section, we've looked at the RPC-style, which is very close to standard MVC Controller actions. In the following sections, you learn about a completely different approach, which is based on resources and resource management.

Congratulations; you have finished the implementation of RPC-style and created a beautiful, modern, browser-based game in which two users can play against each other.

Prepare yourself – in the following sections, you're going to look at more advanced techniques and discover how to provide web APIs for interoperability using two of the most famous API communication styles: REST and HATEOAS.

WOW! eBook
www.wowebook.org

To play the game, you can either use two separate private browser windows or use two distinct browsers, such as Chrome, Edge, or Firefox. To test your web APIs, it is advised that you install and use Postman (`https://www.getpostman.com/`), but you can also use any other HTTP REST-compatible client, such as Fiddler (`https://www.telerik.com/fiddler`), SoapUI (`https://www.soapui.org/downloads/soapui.html`), or even Firefox via its advanced features.

Building REST-style web APIs

The REST style was invented by Roy Fielding in the 2000s and is one of the best ways to provide interoperability between systems that are based on multiple technologies, whether it be in your network or on the internet.

Furthermore, the REST approach is not a technology by itself, but some best practices that are used for efficiently using the HTTP protocol.

Instead of adding a new layer, like SOAP or XML-RPC does, REST uses different elements of the HTTP protocol for providing its services:

- The URI identifies a resource.
- The HTTP verb identifies an action.
- The response is not the resource, but a representation of the resource.
- The client authentication is passed as a parameter in the header of requests.

Unlike the RPC style, the main purpose is no longer to provide actions and is to manage and manipulate resources.

To find out even more about the concepts and ideas behind REST, you should read Roy Fielding's dissertation on this subject, which you can find at `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

As shown in the following diagram, there are mainly three types of resources in the Tic-Tac-Toe application:

- **Users**
- **Game invitations**
- **Game sessions**:

WOW! eBook
www.wowebook.org

Let's learn how to use the REST style for building a game invitation with the REST API:

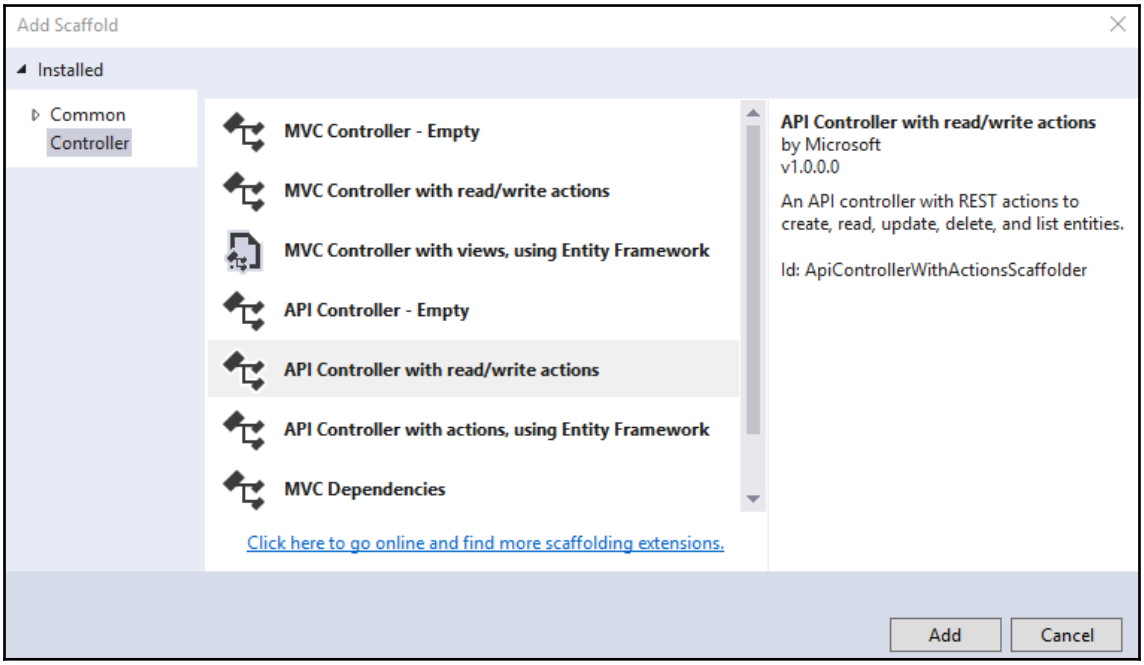
1. Add two new methods, one called `All`, which returns all game invitations, and another called `Delete`, which deletes a game invitation according to the specified game invitation ID. You need to add these two methods to `GameInvitationService` and update the game invitation service interface accordingly:

```
public Task<IEnumerable<GameInvitationModel>> All()
{
  return Task.FromResult<IEnumerable<GameInvitationModel>>
   (_gameInvitations.ToList());
}

public Task Delete(Guid id)
{
  _gameInvitations = new ConcurrentBag<GameInvitationModel>
```


WOW! eBook
www.wowebook.org

```
            (_gameInvitations.Where(x => x.Id != id));
          return Task.CompletedTask;
        }
```

2. Add a new API controller called `GameInvitationApiController`, right-click on the `Controllers` folder, and select **Add | Controller**. Then, choose the **API Controller with read/write actions** template:



3. Remove the auto-generated code and replace it with the following REST API implementation:
    1. First, insert the following code as a scaffold for the game invitation API controller, where we have the decorators of expected output and the actual endpoint route. Then, we have a constructor that injects the game invitations service and the user service into the controller, as follows:

```
[Produces("application/json")]
[Route("restapi/v1/GameInvitation")]
public class GameInvitationApiController : Controller
{
  private IGameInvitationService
   _gameInvitationService;
```


WOW! eBook
www.wowebook.org

```
    private IUserService _userService;
    public GameInvitationApiController
      (IGameInvitationService
     gameInvitationService, IUserService userService)
    {
      _gameInvitationService = gameInvitationService;
      _userService = userService;
    }
      ...
  }
```

2. Add the following two `Get` implementation methods. The first returns all the game invitation services, while the other returns a game invitation service according to the ID being specified as `Guid`:

```
[HttpGet]
public async Task<IEnumerable<GameInvitationModel>> Get()
{
  return await _gameInvitationService.All();
}

[HttpGet("{id}", Name = "Get")]
public async Task<GameInvitationModel> Get(Guid id)
{
  return await _gameInvitationService.Get(id);
}
```

3. Add a method that will be used for the creation or insertion of a game invitation, as follows:

```
[HttpPost]
  public IActionResult Post([FromBody]GameInvitationModel
   invitation)
  {
    if (!ModelState.IsValid)
      return BadRequest(ModelState);

    var invitedPlayer =
      _userService.GetUserByEmail(invitation.EmailTo);
    if (invitedPlayer == null) return BadRequest();

      _gameInvitationService.Add(invitation);
      return Ok();
  }
```


WOW! eBook
www.wowebook.org

4. Add the following method, which will be used for updating a game invitation:

```
[HttpPut("{id}")]
          public IActionResult Put(Guid id,
           [FromBody]GameInvitationModel invitation)
          {
            if (!ModelState.IsValid)
              return BadRequest(ModelState);

            var invitedPlayer =
_userService.GetUserByEmail(invitation.EmailTo);
            if (invitedPlayer == null) return BadRequest();

            _gameInvitationService.Update(invitation);
            return Ok();
          }
```

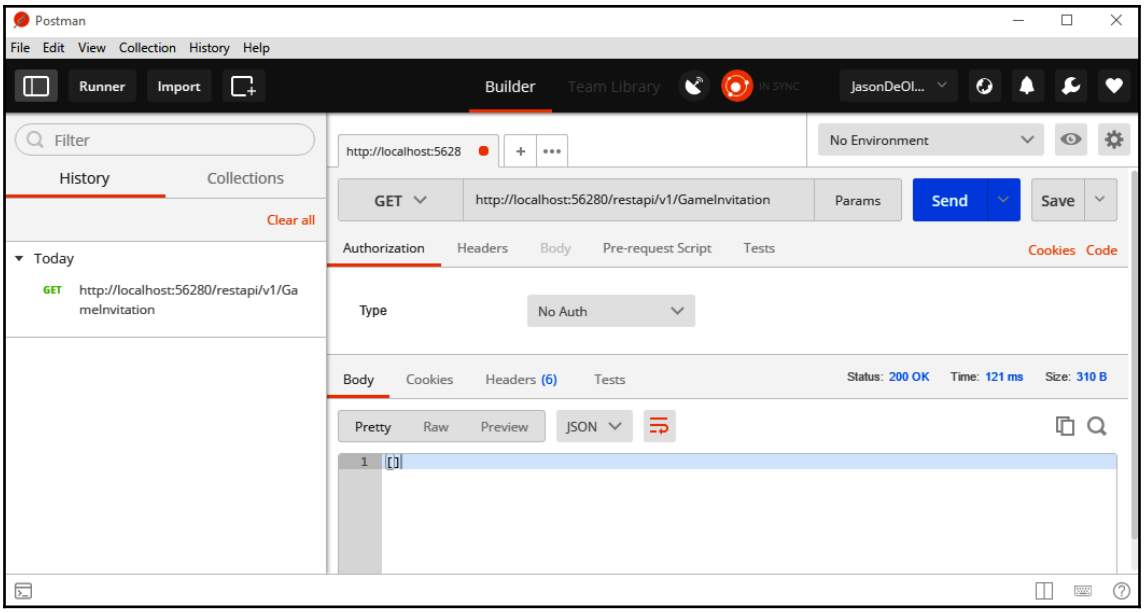
5. Finally, we need to add our delete functionality so that we can delete a game invitation service according to the ID specified as being `Guid`:

```
[HttpDelete("{id}")]
        public void Delete(Guid id)
        {
          _gameInvitationService.Delete(id);
        }
```

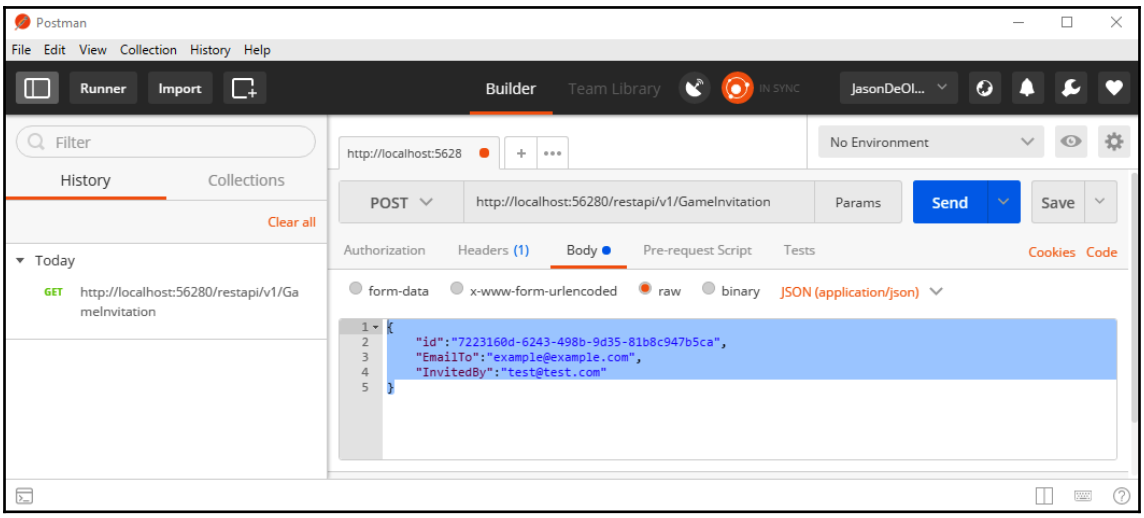
Note that, for learning purposes, we have just provided a very basic example of what you could implement. Normally, you should provide the same functionalities as in your controller implementations (sending emails, confirming emails, verifying data, and so on) and some advanced error handling.

4. Start the application, install and start Postman so that you can do some manual tests on the new REST API you are providing, and send an HTTP `GET` request to `http://<yourhost>/restapi/v1/GameInvitation`. There will be no game invitations since you haven't created any yet:

WOW! eBook
www.wowebook.org



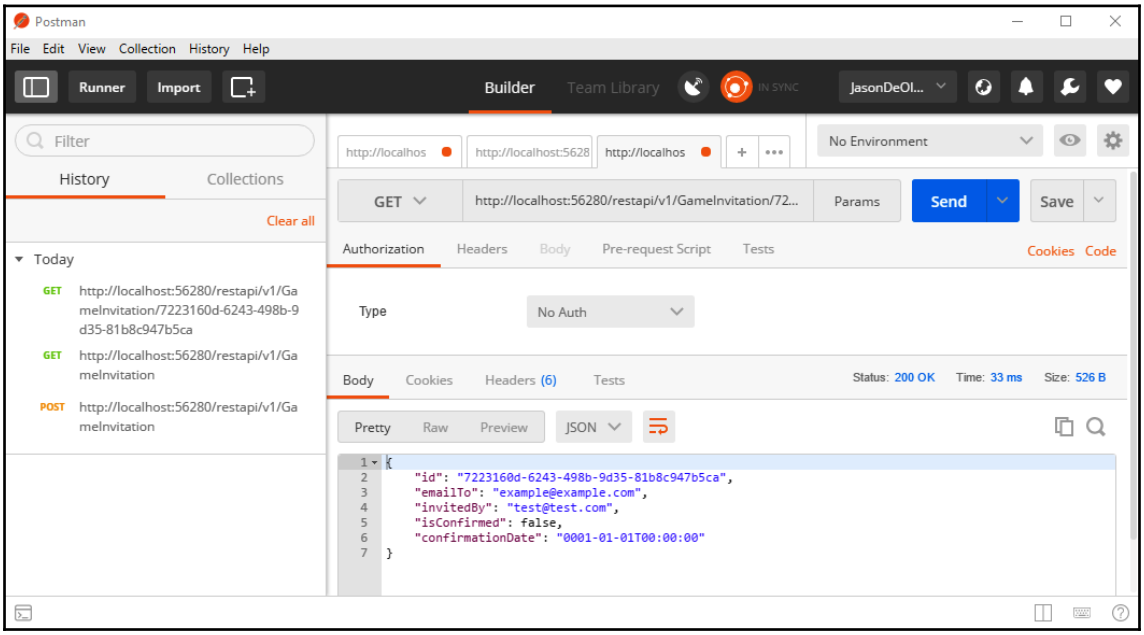
5. Create a new game invitation, send an HTTP `POST` request to `http://<yourhost>/restapi/v1/GameInvitation`, click on **Body**, select **raw** and **JSON**, and use `"id":"7223160d-6243-498b-9d35-81b8c947b5ca"`, `"EmailTo":"example@example.com"`, and `"InvitedBy":"test@test.com"` as parameters:



WOW! eBook
www.wowebook.org

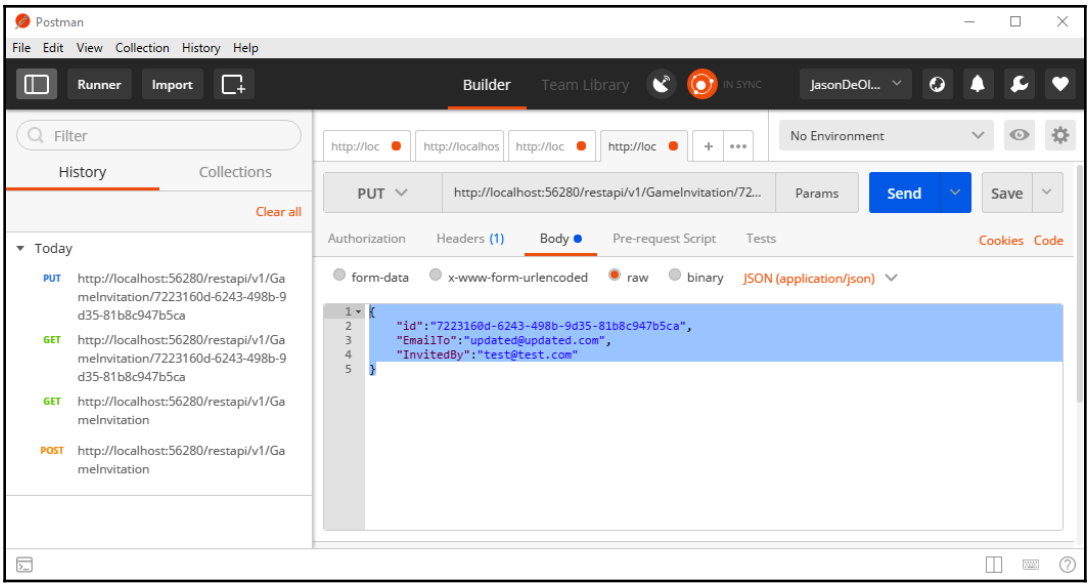
Note that we have added the automatic creation of a user if one doesn't exist for testing purposes in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*. In a real-world scenario, you will have to implement the user registration web APIs and call them before the game invitation web APIs. If you don't, you'll get a bad request since we have added some code to ensure data coherence and integrity.

6. You can retrieve the game invitation either by sending an HTTP `GET` request to `http://<yourhost>/restapi/v1/GameInvitation` or, more specifically, by sending an HTTP `GET` request to `http://<yourhost>/restapi/v1/GameInvitation/7223160d-6243-498b-9d35-81b8c947b5ca`:

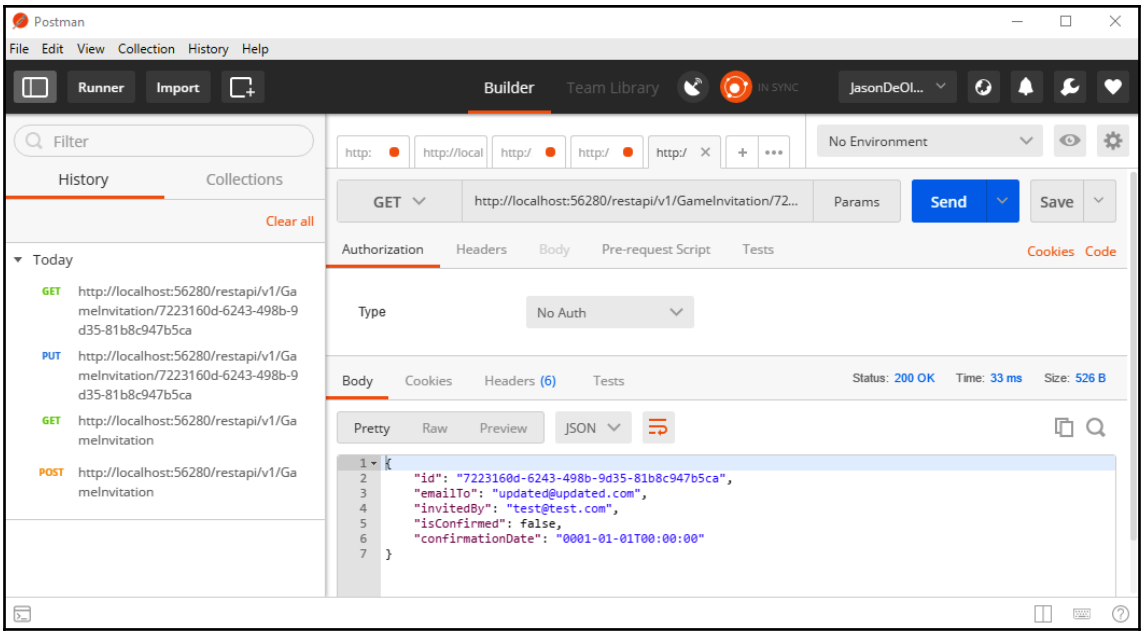


7. Update the game invitation, send an HTTP `PUT` request to `http://<yourhost>/restapi/v1/GameInvitation/7223160d-6243-498b-9d35-81b8c947b5ca`, click on **Body**, select **raw** and **JSON**, and use `"id":"7223160d-6243-498b-9d35-81b8c947b5ca"`, `"EmailTo":"updated@updated.com"`, and `"InvitedBy":"test@test.com"` as parameters:

WOW! eBook
www.wowebook.org

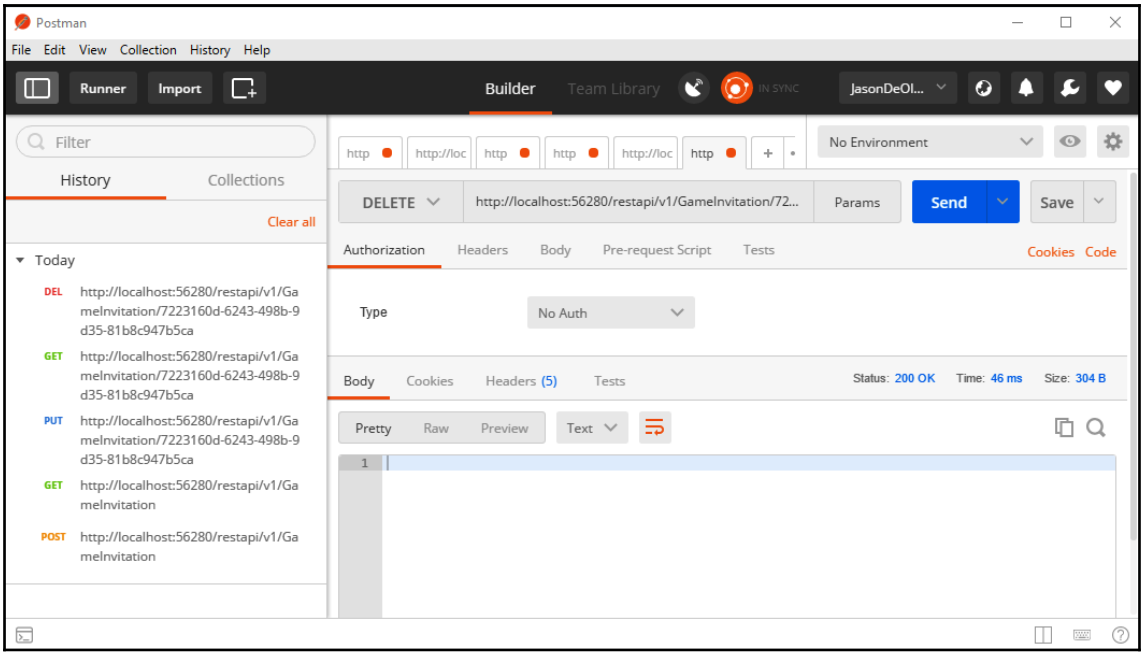


8. Look at the updated game invitation and send an HTTP `GET` request to `http://<yourhost>/restapi/v1/GameInvitation/7223160d-6243-498b-9d35-81b8c947b5ca`:



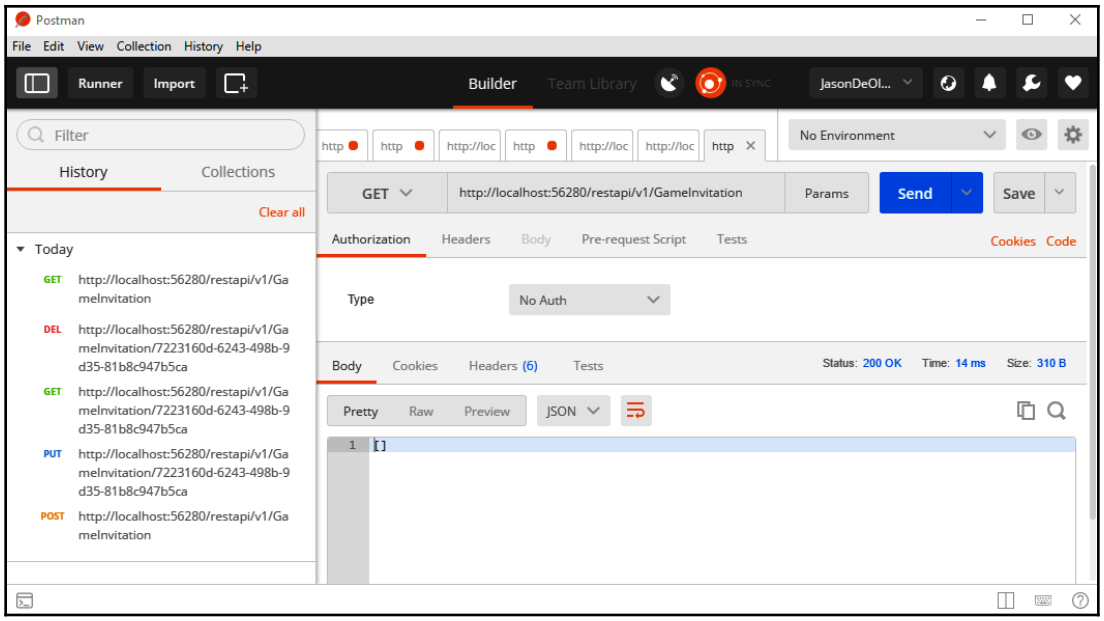
WOW! eBook
www.wowebook.org

9. Delete the game invitation and send an HTTP `DELETE` request to `http://<yourhost>/restapi/v1/GameInvitation/7223160d-6243-498b-9d35-81b8c947b5ca`:



10. Verify the game invitation's deletion and send an HTTP `GET` request to `http://<yourhost>/restapi/v1/GameInvitation`:

WOW! eBook
www.wowebook.org



The REST style is the most common style of web APIs you can find on the market today. It is easy to understand and has been adapted for interoperability use cases.

In the next section, you will learn about a more advanced style called HATEOAS, which is especially well suited for constantly evolving web APIs.

Building HATEOAS-style web APIs

The **Hypermedia as the Engine of Application State** (**HATEOS**) style is yet another approach for providing efficient web APIs. It is, however, completely different from the other two styles we've presented. With this approach, clients can dynamically navigate to a resource by traversing various hypermedia links, which are provided in the HTTP responses.

The advantage of this style is that the server doesn't drive the application state anymore; instead, it is the hypermedia links that are returned by the server that oversee this.

Additionally, compared to the other styles, API changes are handled much better since clients don't hardcode URIs to actions (RPC-style) or resources (REST-style) anymore. Instead, they can work with hypermedia links that have been returned by a server for every response that is received after a request is made. This is an interesting concept in the way that it allows for more flexible and evolvable web APIs.

WOW! eBook
www.wowebook.org

The following diagram shows an example of how to apply the HATEOAS-style to the Tic-Tac-Toe application:

An example of the JSON representation of this diagram is as follows:

```
{
  "_links": {
    "self": { "href": "/gameinvitations" },
    "next": { "href": "/gameinvitations?page=2" },
    "find": {
      "href": "/gameinvitations{?Id}",
      "templated": "true"
    }
  },
  "_embedded": {
    "gameinvitations": [
      {
        "_links": {
          "self": { "href": "/gameinvitations/f1eaf6ac-c998-40da-
            8eb5-198eaa2cc96f" },
          "confirm": { "href": "/gameinvitations/f1eaf6ac-c998-
            40da-8eb5-198eaa2cc96f/confirm" }
        },
        "isConfirmed": "false",
        "confirmDate": "null",
        "emailTo": {
          "self": { "href": "/user/1" }
        },
        "invitedBy": { "self": "\"{\"href\":\"/user/2\"}" }
      }
    ]
  }
}
```


WOW! eBook
www.wowebook.org

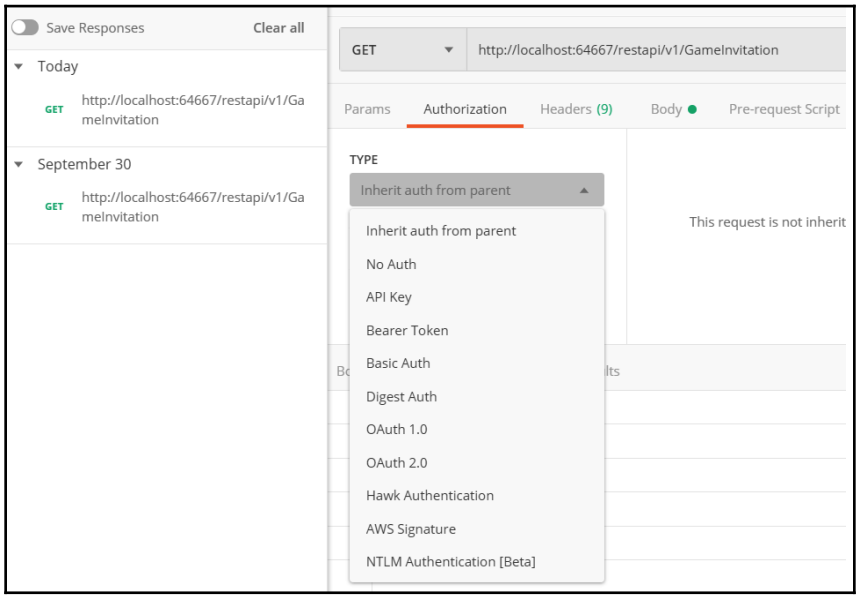
HATEOAS provides some powerful features, all of which allow us to evolve components independently. Clients can be completely decoupled from the business workflows running on the server, which manage interaction by using links and other hypermedia artifacts, such as forms.

Whatever style you use, whether that be RPC, RESTful, or HATEOAS, according to what works best for what scenario and however elegant it is as a solution, it won't be very useful unless your APIs are secure. In the next section, you'll learn about the basics of security for your web APIs.

Securing your web API

At this point, we have managed to create a few API endpoints, but there is a concern that anyone can hit the endpoints from any browser and even manage to modify/delete our game invitations, as long as they know what parameters to pass on. This is a security threat, and you can imagine the implications with an application handling a high level of sensitive functionality.

We will deal with security for ASP.NET Core 3 in `Chapter 10`, *Securing ASP.NET Core 3 Applications*, and `Chapter 11`, *Securing ASP.NET Applications – Vulnerabilities*, but it is worth noting the available security measures for our web API endpoints. Let's have a look at the following screenshot, which shows the **Authorization** tab of Postman:



WOW! eBook
www.wowebook.org

Take note of the different types of authorization that Postman expects, including **No Auth**, meaning no authorization at all.

`Chapter 11`, *Securing ASP.NET Applications – Vulnerabilities*, will give us insight into the common security vulnerabilities that we have to watch out for. With this in mind, it is always important to secure our web API endpoints with any of the following authorization options:

- API key
- Bearer token
- Basic auth
- Digest auth
- OAuth 1.0
- OAuth 2.0
- Hawk authentication
- AWS signature
- NTLM authentication

These authentication options are explained further on the following documentation, which talks about authorization in Postman: `https://learning.getpostman.com/docs/postman/sending-api-requests/authorization/`.

Apart from making our APIs secure from unwanted users, there are legitimate users that we need to make sure have a great experience using our APIs. One of the ways of helping our users do this is by giving them access to documentation using our API specifications. We will learn how to do this in the next section.

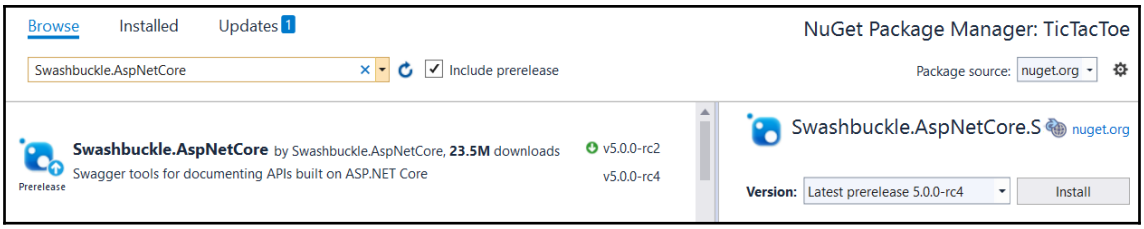
ASP.NET Core web API help pages with Swagger/OpenAPI

As the size of any application grows, and the number of web API endpoints grows, it is often a good idea to have documentation about the API itself, the available endpoints, what they expect as parameters, and what to expect as a normal response from any respective API calls that are made.

WOW! eBook
www.wowebook.org

It can be tedious to document every API endpoint manually, but fortunately, Swagger/OpenAPI comes to the rescue here. Let's take a look:

1. Go to our Tic-Tac-Toe demo application, right-click on the `TicTacToe` project, go to the NuGet Package Manager, and search for `Swashbuckle.AspnetCore`. Now, click on the **Install** button:



You can also install Swashbuckle by going to the Package Manager Console and typing `Install-Package Swashbuckle.AspNetCore -Version 5.0.0-rc4` at the Package Manager's Command Prompt.

2. Next, add the following code snippet to the `ConfigureServices` method in the `Startup` class:

```
services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo {
        Title = "Learning ASP.Net Core 3.0 Rest-API",
        Version = "v1",
        Description = "Demonstrating auto-generated
         API documentation",
        Contact = new OpenApiContact
        {
            Name = "Kenneth Fukizi",
            Email = "example@example.com",
        },
        License = new OpenApiLicense
        {
            Name = "MIT",
        }
    });
```

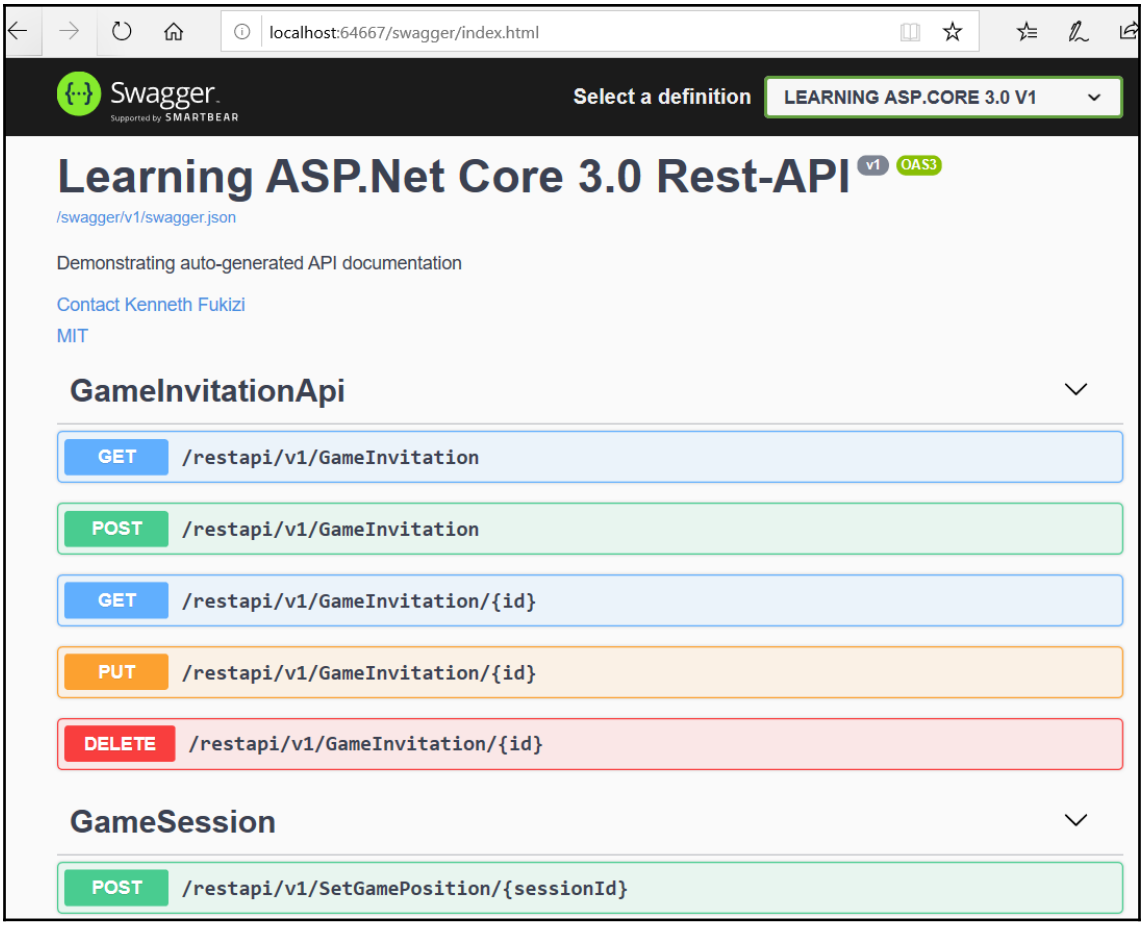
Make sure that you import the `using Microsoft.OpenApi.Models;` namespace so that you can use the `OpenApiInfo` class.

WOW! eBook
www.wowebook.org

3. Finally, we need to add the following code to the `Configure` method in the same `Startup.cs` class:

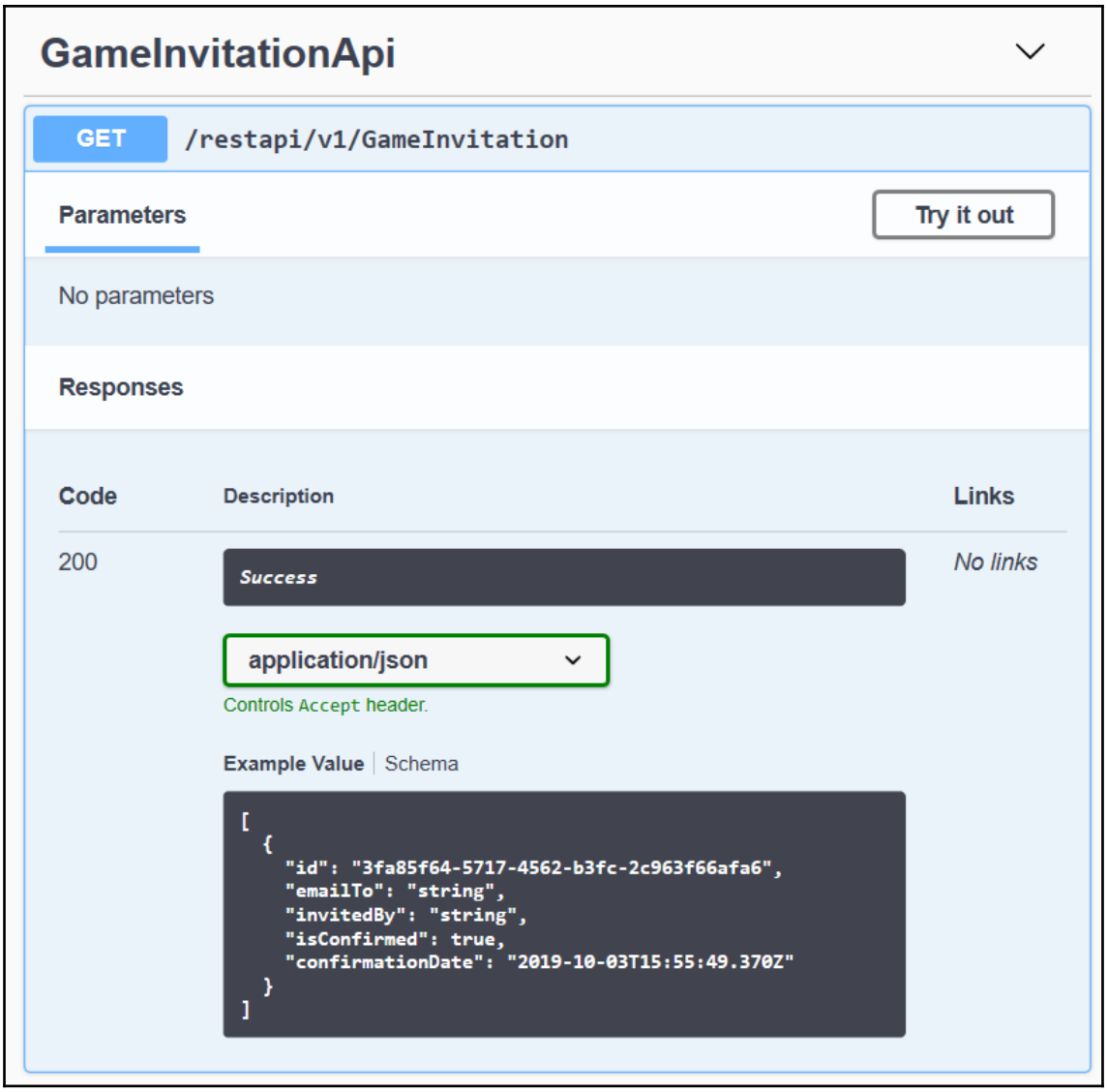
```
app.UseSwagger();

app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json",
     "LEARNING ASP.CORE 3.0 V1");
});
```

4. Start the `TicTacToe` demo application and add Swagger to the root URL. You will see all the API endpoints we have created so far, all of which are documented on the Swagger index page:



WOW! eBook
www.wowebook.org

5. Swagger can also be used to test out the intended functionality for any API endpoint instead of the other most common tools, such as Postman and Fiddler. Of course, another common way to test API endpoints is by typing them in manually as browser URLs. You can test an endpoint using Swagger by clicking on it (to expand it), clicking on the **Try it out** button, as shown to the right of the following screenshot, and entering the expected values:



WOW! eBook
www.wowebook.org

6. We may want to have the API documentation on the main index page, especially in cases where the whole application is an API that we are developing for other users. In this case, all we need to do is add an empty `RoutePrefix` `SwaggerUIOption`, as follows:

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json",
     "LEARNING ASP.CORE 3.0 V1");
    c.RoutePrefix = string.Empty;
});
```

For every additional API endpoint you wish to add, Swagger will pick it up and document it automatically, leaving you to concentrate on just producing the code and not the documentation – isn't that liberating?

Summary

In this chapter, you have learned how to build web APIs for your applications for integration purposes and for loosely coupled application architectures.

We have explored different three styles for our web APIs, that is, RPC, REST, and HATEOAS. Each of those styles has specific advantages and use cases. You have to choose carefully, depending on your specific application needs, since there is no one single style that outclasses the others.

Throughout this chapter, we've looked at examples of how to transform existing controller actions into RPC-style web APIs and how to build REST-style and HATEOAS-style web APIs from the ground up. Then, we used Postman to manually test our web APIs and you have acquired enough knowledge to apply all of these new concepts to your own environments. Finally, we used OpenAPI's Swagger to automatically produce documentation for our API endpoints.

To summarize, we've learned how to build REST APIs, acquired skills in terms of how to transform a controller action into an RPC-style web API, and built them from scratch. We also learned how to build HATEOAS-style web APIs and configure our APIs so that they have a help page with API specification documentation.

In the next chapter, we will talk about how to access data by using Entity Framework Core 3 in our ASP.NET Core 3 applications.

WOW! eBook
www.wowebook.org

3
Section 3: The ASP.NET Core 3 Supporting Ecosystem

In this section, we will walk you through the process of persisting data through the use of Entity Framework Core 3 and retrieving it when needed. Then, we will deal with the issues of hosting deployed applications and take you through the process of monitoring them. Security will be talked about in detail in a chapter dedicated to it in order to underline how important it is to make sure that applications are safe.

This section comprises the following chapters:

- `Chapter 9`, *Accessing Data Using Entity Framework Core 3*
- `Chapter 10`, *Securing ASP.NET Core 3 Applications*
- `Chapter 11`, *Securing ASP.NET Applications – Vulnerabilities*
- `Chapter 12`, *Hosting ASP.NET Core 3 Applications*
- `Chapter 13`, *Managing ASP.NET Core 3 Applications*

WOW! eBook
www.wowebook.org

9
Accessing Data Using Entity Framework Core 3

We have come a long way with our implementation of the Tic-Tac-Toe demo web application, but when we restart the application, none of our user registration and application data is remembered. This is due to the fact that we are **not saving or persisting any data yet**.

To persist data and be able to reload it when the application starts, we have to put it into some kind of persistent storage, such as files (XML, JSON, CSV) or databases.

A database would be the best choice since it provides better performance and more security compared to simple file storage, and that is why we are going to use this approach in the examples in this chapter.

Since the old ASP.NET 3 days, we have been able to use an **object-relational mapping** (**ORM**) framework called **Entity Framework** to access data in databases in a more productive and simple way. ASP.NET Core 3 works seamlessly with a dedicated version of this framework called **Entity Framework Core 3**, part of Entity Framework 6.3, and can also work with previous versions.

We will start this chapter by introducing Entity Framework Core 3.0 and how to install it. Then, we will learn about all the classes we need in order to use a code-first approach to create the database, after which we will show you how to perform migrations. Next, we'll explore normal CRUD operations and explain the most common and important data relationships. Finally, we will explain queries in slightly greater depth and introduce transactions.

By the end of this chapter, you will be able to connect to a database using Entity Framework Core, use migrations with updates, carry out basic CRUD operations, work with the Fluent API, perform complex queries against a database, and use transactions.

WOW! eBook
www.wowebook.org

In this chapter, we will cover the following topics:

- Getting started with Entity Framework Core 3
- Working with Entity Framework Core 3 Data Annotations
- Using Entity Framework Core 3 migrations
- Creating, reading, updating, and deleting data
- Understanding data relationships
- Working with queries
- Using transactions

Getting started with Entity Framework Core 3

The `Microsoft.AspNetCore.App` meta-package contains Entity Framework Core 3, including all the packages you need to work with **Microsoft SQL Server** and **SQLite**.

Note that, if you need to work with other databases such as MySQL, you have to download additional packages from NuGet.

You can find a list of all the currently available Entity Framework Core 3 NuGet packages here: `https://www.nuget.org/packages?page=2&q=Tags%3A%22entity-framework-core%22`.

Entity Framework is Microsoft's version of an ORM, and is not the only one that can be used on ASP.NET Core. Other ORMs of note that work seamlessly with .NET Core include NHibernate, LINQ to SQL, and Dapper.

ORMs are the recommended way to access databases, especially **relational database management systems** (**RDBMS**), in order to counteract the well documented impedance mismatch. ORMs abstract you, as a developer, away from the nitty gritty of SQL manipulations and implementations.

Entity Framework Core 3.0 is a later version of Entity Framework Core 1.0 and subsequent versions that have been evolving since Entity Framework versions were specifically designed for the .NET Framework.

WOW! eBook
www.wowebook.org

You can install Entity Framework Core 3.0 by running the following command on the Package Manager Console:

```
Install-package Microsoft.EntityFrameworkCore
```

By running the preceding command, you will receive the following output:

```
Package Manager Console
Package source: All        Default project: EntityFramework

PM> Install-package Microsoft.EntityFrameworkCore
Restoring packages for C:\Projects\PacktPublishing\EntityFramework\EntityFramework.csproj...
Installing NuGet package Microsoft.EntityFrameworkCore 3.0.0.
Committing restore...
Writing assets file to disk. Path: C:\Projects\PacktPublishing\EntityFramework\obj\project.assets.json
Restore completed in 286.36 ms for C:\Projects\PacktPublishing\EntityFramework\EntityFramework.csproj.
Successfully installed 'Microsoft.EntityFrameworkCore 3.0.0' to EntityFramework
Successfully installed 'Microsoft.EntityFrameworkCore.Abstractions 3.0.0' to EntityFramework
Successfully installed 'Microsoft.EntityFrameworkCore.Analyzers 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Caching.Abstractions 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Caching.Memory 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Configuration 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Configuration.Abstractions 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Configuration.Binder 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.DependencyInjection 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.DependencyInjection.Abstractions 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Logging 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Logging.Abstractions 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Options 3.0.0' to EntityFramework
Successfully installed 'Microsoft.Extensions.Primitives 3.0.0' to EntityFramework
Successfully installed 'System.Collections.Immutable 1.6.0' to EntityFramework
Successfully installed 'System.ComponentModel.Annotations 4.6.0' to EntityFramework
Successfully installed 'System.Diagnostics.DiagnosticSource 4.6.0' to EntityFramework
Successfully installed 'System.Threading.Tasks.Extensions 4.5.2' to EntityFramework
Executing nuget actions took 876.33 ms
Time Elapsed: 00:00:01.6410634
PM>
```

You will also need to install the SQL Server provider with the following command on the Package Manager Console since they work hand in hand:

```
Install-package Microsoft.EntityFrameworkCore.SqlServer
```

Before you actually start to use EF Core 3.0, it is only natural to try to establish a connection to the database first, which we will look at in the next section.

WOW! eBook
www.wowebook.org

Establishing a connection

To open a session to the database and query and update instances of your entities, you need to use `DbContext`, which is based on a combination of the unit of work and repository patterns.

Let's learn how to prepare the Tic-Tac-Toe application so that we can use Entity Framework Core 3 from scratch to connect to a SQL database via `DbContext` and a connection string:

1. Go to the Solution Explorer and add a new folder called `Data`, add a new class called `GameDbContext`, and implement a `DbSet` property for each model (`UserModel`, `TurnModel`, and so on):

```
public class GameDbContext : DbContext
{
   public DbSet<GameInvitationModel> GameInvitationModels
    {get; set; }
   public DbSet<GameSessionModel> GameSessionModels { get;
    set; }
   public DbSet<TurnModel> TurnModels { get; set; }
   public DbSet<UserModel> UserModels { get; set; }
   public GameDbContext(DbContextOptions<GameDbContext>
    dbContextOptions) :      base(dbContextOptions)  { }
}
```

2. Register `GameDbContext` in the `Startup` class. Then, pass the connection string and database provider as parameters within the constructor. Currently, we only need a single instance, so we will use `AddSingleton`:

```
var connectionString =
  _configuration.GetConnectionString("DefaultConnection");
services.AddEntityFrameworkSqlServer()
  .AddDbContext<GameDbContext>((serviceProvider,
   options) => options.UseSqlServer(connectionString).
    UseInternalServiceProvider(serviceProvider)
);

var dbContextOptionsbuilder =
  new DbContextOptionsBuilder<GameDbContext>()
  .UseSqlServer(connectionString);
services.AddSingleton(dbContextOptionsbuilder.Options);
```

Please note that you will need to add the following `using` statements for the code to compile: `using TicTacToe.Data;` and `using Microsoft.EntityFrameworkCore;`.

WOW! eBook
www.wowebook.org

3. Update the user service class called `UserService.cs` so that you can work with the game database context: `GameDbContext.cs`. Add a new public constructor and a private member for the game database context:

```
private DbContextOptions<GameDbContext> _dbContextOptions;
public UserService(DbContextOptions<GameDbContext>
 dbContextOptions)
{
  _dbContextOptions = dbContextOptions;
}
```

4. Update the `RegisterUser` method in `UserService` so that you can use the game database context: `GameDbContext`:

```
public async Task<bool> RegisterUser(UserModel userModel)
{
  using(var Database = new GameDbContext
    (_dbContextOptions))
  {
    Database.UserModels.Add(userModel);
    await Database.SaveChangesAsync();
    return true;
  }
}
```

5. Add a new extension called `ModelBuilderExtensions` to the `Extensions` folder. This will be used to define table name conventions:

```
public static class ModelBuilderExtensions
{
  public static void RemovePluralizingTableNameConvention(
   this ModelBuilder modelBuilder)
  {
    foreach (IMutableEntityType entity in
     modelBuilder.Model.GetEntityTypes())
    {
      entity.SetTableName(entity.DisplayName());
    }
  }
}
```


WOW! eBook
www.wowebook.org

6. Update the `OnModelCreating` method in the game database context, `GameDbContext`, to configure the models to configure the models that were discovered from the entity types exposed in `DbSet` properties, for example: `public DbSet<UserModel> UserModels { get; set; }`. Then, we call the `ModelBuilderExtensions` extension class to apply the table name conventions:

```
protected override void OnModelCreating(ModelBuilder
 modelBuilder)
{
  modelBuilder.RemovePluralizingTableNameConvention();
}
```

Note that we could also use another method called `OnConfiguring` in the database context in order to configure the database context without using `DbContextOptions`.

7. Add a new class called `GameDbContextFactory` to the `Data` folder. This will be used to instantiate the game database context, `GameDbContext`, with specific options:

```
public class GameDbContextFactory :
 IDesignTimeDbContextFactory<GameDbContext>
{
  public GameDbContext CreateDbContext(string[] args)
  {
    var optionsBuilder =  new
        DbContextOptionsBuilder<GameDbContext>();
    optionsBuilder.UseSqlServer(@"Server=
     (localdb)\MSSQLLocalDB;Database=TicTacToe;
     Trusted_Connection=True;
       MultipleActiveResultSets=true");
    return new GameDbContext(optionsBuilder.Options);
  }
}
```

Note that you will have to add the following `using` statements so that the code compiles: `using Microsoft.EntityFrameworkCore;` and `using Microsoft.EntityFrameworkCore.Design;`.

If you have worked with databases before, you should be familiar with the concept of connection strings. They contain the configuration (address, username, password, and more) and settings (encryption, protocol, and more) that are required so that we can connect to a database.

WOW! eBook
www.wowebook.org

In ASP.NET Core 3, you can also use an `appSettings.<env>.json` file to configure connection strings. Connection strings are loaded automatically when we use the `ConnectionStrings` section within this file:

```
"ConnectionStrings": {
  "DefaultConnection":
  "Server=(localdb)\\MSSQLLocalDB;Database=TicTacToe;
    Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

As you can see, you can use the `GetConnectionString` method to retrieve a connection string at runtime:

```
var databaseConnectionString =
  _configuration.GetConnectionString("DefaultConnection");
```

This is everything you need to know in order to use the game database context, `GameDbContext`, and the corresponding default connection string stored within the `appsettings.json` configuration file of the Tic-Tac-Toe application.

Before we start using the game database context on our models, it is important to make sure that we have grasped all the required basics so that we can use our game database context to create and access database tables. It is recommended that every database table has a primary key if it is to qualify as a relational table; it needs to have a foreign key if it is related to another table. In the next section, we will look at how we are going to define our primary and foreign keys in our code through Data Annotations.

Defining primary keys and foreign keys via Data Annotations

Now, we need to modify existing models so we can persist them within a SQL database. To allow Entity Framework Core 3.0 to create, read, update, and delete records, we need to specify a primary key for each model. We can do this by using Data Annotations, which allow us to decorate a property with the `[Key]` decorator.

The following is an example of how to use Data Annotations for `UserModel`:

```
public class UserModel
{
  [Key]
  public long Id { get; set; }
  ...
}
```


WOW! eBook
www.wowebook.org

You should apply this to `UserModel`, `GameInvitationModel`, `GameSessionModel`, and `TurnModel` in the Tic-Tac-Toe application. You can reuse existing `Id` properties and decorate them with the `[Key]` decorator, or add new ones if a model doesn't contain an `Id` property yet.

Note that it is sometimes required to use composite keys as the identity for your rows in a table. In this case, decorate each property with the `[Key]` decorator. Furthermore, you can use `Column[Order=]` to define the position of the property if you need to order a composite key.

When working with **SQL Server** (or any other SQL 92 DBMS), the first thing you should think about is the relationship between tables. In Entity Framework Core 3, you can specify foreign keys within models by using the `[ForeignKey]` decorator.

Concerning the Tic-Tac-Toe application, this means that you have to update `GameInvitationModel` and add a foreign key relationship to the user model ID. Perform the following steps to do so:

1. Update `GameInvitationModel` and add a foreign key attribute to the `InvitedByUser` property:

```
public class GameInvitationModel
{
  [Key]
  public Guid Id { get; set; }
  public string EmailTo { get; set; }

  public string InvitedBy { get; set; }
  public UserModel InvitedByUser {get; set;}
  [ForeignKey(nameof(InvitedByUserId))]
  public Guid InvitedByUserId { get; set; }

  public bool IsConfirmed { get; set; }
  public DateTime ConfirmationDate { get; set; }
}
```

This is an already existing `GameInvitationalModel` class, and we are just decorating the property `Id` with a `[Key]` attribute so that Entity Framework Core 3.0 will be able to identify it as a primary key. The foreign key attribute, `[ForeignKey(nameof(InvitedByUserId)]`, decorates the GUID called `InvitedUserId` so that EF Core 3.0 will be able to see this property as a foreign key to another table.

WOW! eBook
www.wowebook.org

2. Update `GameSessionModel` and add a foreign key to `UserId1`:

```
public class GameSessionModel
{
  [Key]
  public Guid Id { get; set; }
  ...
  [ForeignKey(nameof(UserId1))]
  public UserModel User1 { get; set; }
  ...
}
```

Here, we have a `GameSessionModel` POCO class that's decorated with a primary key attribute on its `Id` property and a secondary key attribute on the user model called `User 1`. This will allow EF Core 3.0 to create a `GameSessionModel` table with a primary key called `Id` and a foreign key called `User1`, respectively.

3. Update `TurnModel` and add a foreign key to `UserId`:

```
public class TurnModel
{
  [Key]
  public Guid Id { get; set; }
  [ForeignKey(nameof(UserId))]
  public Guid UserId { get; set; }
  public UserModel User { get; set; }
  public int X { get; set; }
  public int Y { get; set; }
  public string Email { get; set; }
  public string IconNumber { get; set; }
}
```

Entity Framework Core 3 maps all properties in a model with a schema representation by default. But some more complex property types are not compatible, which is why we should exclude them from auto-mapping. But how do we do this? Well, by using the `[NotMapped]` decorator. How easy and straightforward is that?

4. For the Tic-Tac-Toe application, it makes no sense to persist the active user for a turn, so you should exclude this from the auto-mapping process by using the `[NotMapped]` decorator in `GameSessionModel`:

```
public class GameSessionModel
{
  [Key]
  public Guid Id { get; set; }
```


WOW! eBook
www.wowebook.org

```
    ...

    [NotMapped]
    public UserModel Winner { get; set; }

    [NotMapped]
    public UserModel ActiveUser { get; set; }
    public Guid WinnerId { get; set; }
    public Guid ActiveUserId { get; set; }
    public bool TurnFinished { get; set; }
    public int TurnNumber { get; set; }
}
```

Now that you have decorated all your models using Entity Framework Core 3 Data Annotations, you will notice that you have two properties, `User1` and `User2`, in `GameSessionModel` that point to the same `UserModel` entity. This results in a circular relationship, and that will give us a problem (when we work with relational databases) to performing operations such as cascading updates or cascading deletions.

For more information on Entity Framework Data Annotations, please visit `https://msdn.microsoft.com/en-us/library/jj591583(v=vs.113).aspx`.

5. To avoid circular relationships, you need to decorate `User1` with the `[ForeignKey]` decorator and update the `OnModelCreating` method in the game database context, `GameDbContext`, to define the foreign key for `User2`. These two modifications will allow you to define the two foreign keys while avoiding automatic cascading operations, which would cause problems:

```
protected override void OnModelCreating(ModelBuilder
 modelBuilder)
{
  modelBuilder.RemovePluralizingTableNameConvention();
  modelBuilder.Entity(typeof(GameSessionModel))
  .HasOne(typeof(UserModel), "User2")
  .WithMany()
  .HasForeignKey("User2Id").OnDelete(DeleteBehavior.Restrict);
}
```


WOW! eBook
www.wowebook.org

6. Now, you need to fix the unit tests. You may have already noticed that the unit test project doesn't build anymore if you try compiling the solution. Here, you need to update the unit test, since `UserService` now requires an instance of `DbContextOptions`, as follows:

```
var dbContextOptionsBuilder =
  new DbContextOptionsBuilder<GameDbContext>()
  .UseSqlServer(@"Server=
    (localdb)\MSSQLLocalDB;Database=TicTacToe;
   Trusted_Connection=True;MultipleActiveResultSets=true");

var userService = new
  UserService(dbContextOptionsBuilder.Options);
```

Please note that, while the preceding code snippet fixes the tests, it is not good practice to work with real database connections inside unit tests. Ideally, the data connection should be mocked or abstracted in some way. If you need to use real data for integration tests, the connection information should come from a config file instead of being hardcoded.

Now, the unit tests cater for a constructor of `UserService` with the new overload of the options builder. Now that we have defined our primary and foreign keys in our models, we can create our initial database schema and prepare our application for migration. In the next section, we look at EF Core 3 migrations.

Using Entity Framework Core 3 migrations

As you have already seen, when developing applications, your models may change frequently when you refactor and finalize your projects. This may lead to a database schema that is out of sync and that needs to be updated manually. You can do this by creating an upgraded script, but this isn't an ideal solution.

Fortunately, Entity Framework Core 3 includes a feature called **migrations** to help you with this tedious task. It automatically keeps your models and their corresponding database schemas in sync.

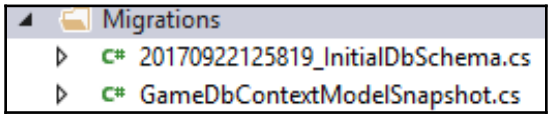
WOW! eBook
www.wowebook.org

After you have updated the models, services, and controllers so that they comply with the preceding constraints and modified the game database context, `GameDbContext`, accordingly, you are ready to use Entity Framework Core 3 migrations. The following steps will show you how to use Entity Framework Core 3 migrations:

1. Add a first version of your database schema called `InitialDbSchema`. To do this, open the **NuGet Package Manager** by clicking on **Tools | NuGet Package Manager | Package Manager Console** and execute the `Add-Migration InitialDbSchema` command:

```
Package Manager Console
Package source: All        Default project: TicTacToe
Each package is licensed to you by its owner. NuGet is not responsible for,
nor does it grant any licenses to, third-party packages. Some packages may include
dependencies which are governed by additional licenses. Follow the package
source (feed) URL to determine any dependencies.

Package Manager Console Host Version 4.4.0.4431

Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration InitialDbSchema
189 %
```

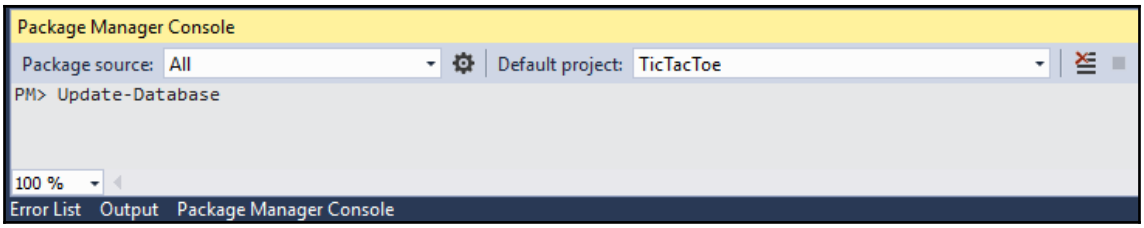
2. A new folder called `Migrations` will be automatically added by Visual Studio. It will contain two autogenerated files that will help you manage and upgrade your database schema in the future:



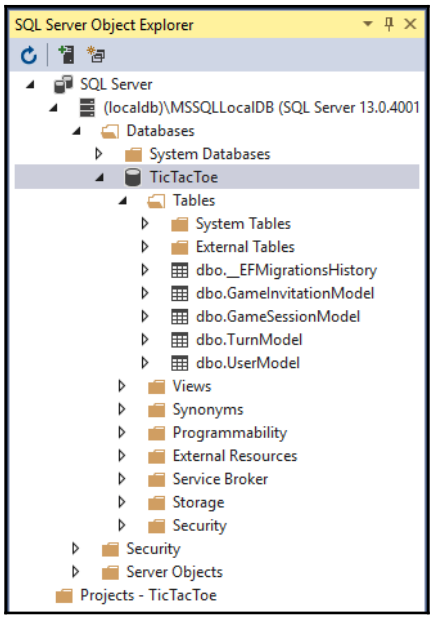
You can update your database directly from within Visual Studio 2019 if it is accessible from your development environment. The following steps will walk you through the update process:

1. Go to the **Package Manager Console** and execute the `Update-Database` command. This will create a database the first time it is used, or update the database automatically when you change your models:

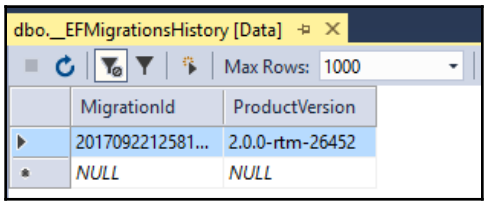
WOW! eBook
www.wowebook.org



2. Then, go to the **SQL Server Object Explorer** and analyze the database schema that Entity Framework 3 migrations has autogenerated in SQL Server:



3. After that, right-click on the `__EFMigrationsHistory` table and select **View Data** to see how Entity Framework migrations tracks database schema versions:



WOW! eBook
www.wowebook.org

If your database is not accessible from your development environment (for example, in staging or production), you have to generate a SQL script file.

4. Go to the **Package Manager Console** and execute the `Script-Migration` command to autogenerate a SQL script file, which can be used to create the Tic-Tac-Toe application's database:

```
denrd0ia.sql
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;

GO

CREATE TABLE [UserModel] (
    [Id] bigint NOT NULL IDENTITY,
    [Email] nvarchar(max) NOT NULL,
    [EmailConfirmationDate] datetime2 NULL,
    [FirstName] nvarchar(max) NOT NULL,
    [IsEmailConfirmed] bit NOT NULL,
    [LastName] nvarchar(max) NOT NULL,
    [Password] nvarchar(max) NOT NULL,
    [Score] int NOT NULL,
    CONSTRAINT [PK_UserModel] PRIMARY KEY ([Id])
);

GO

CREATE TABLE [GameInvitationModel] (
    [Id] uniqueidentifier NOT NULL,
    [ConfirmationDate] datetime2 NOT NULL,
    [EmailToId] bigint NOT NULL,
    [InvitedById] bigint NOT NULL,
    [IsConfirmed] bit NOT NULL,
    CONSTRAINT [PK_GameInvitationModel] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_GameInvitationModel_UserModel_EmailToId] FOREIGN KEY ([EmailToId]) REFERENCES [UserModel] ([Id]) ON DELETE NO ACTION,
    CONSTRAINT [FK_GameInvitationModel_UserModel_InvitedById] FOREIGN KEY ([InvitedById]) REFERENCES [UserModel] ([Id]) ON DELETE CASCADE
);

GO

CREATE TABLE [GameSessionModel] (
    [Id] uniqueidentifier NOT NULL,
    [TurnNumber] int NOT NULL,
    [User1Id] bigint NOT NULL,
    [User2Id] bigint NOT NULL,
    CONSTRAINT [PK_GameSessionModel] PRIMARY KEY ([Id]),

Package Manager Console
Package source: All    Default project: TicTacToe
PM> Script-Migration
100 %
Error List  Output  Package Manager Console
```

5. After that, execute the generated SQL script file on specific environments like staging and production using your preferred database tools (for example, SQL Server Management Studio, and so on) to create the Tic-Tac-Toe application's database.

WOW! eBook
www.wowebook.org

You can also use Entity Framework Core 3 migration directly from within your code to ensure that the database is constantly in sync with your models. To do this, you need to call the `Migrate` method of the `GameDbContext` instance within the `Configure` method of the `Startup` class. Perform the following steps to do so:

1. Update the `Configure` method in the `Startup` class and add the following instructions at the bottom of the method:

```
using (var scope =
 app.ApplicationServices.GetService<IServiceScopeFactory>()
  .CreateScope())
{
  scope.ServiceProvider.GetRequiredService<GameDbContext>()
   .Database.Migrate();
}
```

   This places the game database context's `Migrate` method as a scoped service that can be resolved at application runtime.

2. Start the Tic-Tac-Toe application by pressing *F5*

Note that if a table or property doesn't exist in the database and if the connection string provides enough access rights, Entity Framework Core 3 will automatically create the missing table or the property/column that does not exist.

Now that we've updated the models and the corresponding application database, all the model data will be persisted and the application state is going to be available, even after an application restart. This means that you cannot register already existing emails, you have to add new ones manually, so truncate the database and delete them now.

In the next section, we will focus on creating, reading, updating, and deleting data.

Creating, reading, updating, and deleting data

So far, we have defined our models and got the database up and running in a consistent and coherent way. In this section, we will learn how to work with data and execute create, read, update, and delete operations.

WOW! eBook
www.wowebook.org

Let's learn how to use `GameDbContext` to work with data:

1. First, update `UserService`, remove `ConcurrencyBag` and the static constructor, and update the `GetUserByEmail` method:

```
public async Task<UserModel> GetUserByEmail(string email)
{
  using (var Database = new
    GameDbContext(_dbContextOptions))
  {
    return await Database.UserModels.FirstOrDefaultAsync(
     x => x.Email == email);
  }
}
```

2. Update the `UpdateUser` method in `UserService` to learn how to update data using the database context:

```
public async Task UpdateUser(UserModel userModel)
{
  using (var gameDbContext =
    new GameDbContext(_dbContextOptions))
  {
    gameDbContext.Update(userModel);
    await gameDbContext.SaveChangesAsync();
  }
}
```

3. Update the `GetTopUsers` method within `UserService` to learn how to build advanced queries with sorting and filtered data using the database context:

```
public async Task<IEnumerable<UserModel>> GetTopUsers(
 int numberOfUsers)
{
  using (var gameDbContext =
   new GameDbContext(_dbContextOptions))
  {
    return await
gameDbContext.UserModels.OrderByDescending(
     x => x.Score).ToListAsync();
  }
}
```


WOW! eBook
www.wowebook.org

4. Add a new method called `IsUserExisting` to `UserService`. This will be used to check whether a user exists. Update the `IUserService` interface:

```
public async Task<bool> IsUserExisting(string email)
{
  using (var gameDbContext =
   new GameDbContext(_dbContextOptions))
  {
    return await gameDbContext.UserModels.AnyAsync(
     user => user.Email == email);
  }
}
```

In this section, you've learned how to configure your applications so that they can use Entity Framework Core 3 and all of its useful and interesting features. This is a great way of abstracting complexity and removing time-consuming tasks from your daily life as a developer.

You don't need to learn about any additional languages (such as SQL), nor do you need to change environments to create, read, update, and delete records in a database. Everything can be done from within your code and from within Visual Studio to ensure high developer productivity and efficiency.

Understanding data relationships

Let's take a breather from our normal demo Tic-Tac-Toe game application and look at a few Entity Framework Core 3.0 concepts in a little more detail.

Before you understand and make any advanced queries, it is important to understand what kinds of relationship are possible between two or more entities with respect to data.

We have already looked at the basics in terms of primary keys and foreign keys, but let's go over their definitions, which will help you understand the terms a bit more.

Primary key

A primary key is designed to uniquely identify every record in a table. For example, in a student table, it will be the student ID column. We need to be mindful that a primary key can be a composite key, in which two different columns can be combined into a unique identifier of records within a table.

WOW! eBook
www.wowebook.org

Within any table, every record must have a non-empty value in the primary key column, and it should always be unique and never repeated.

For relational tables, it is imperative to have a primary key defined. After its definition, you cannot define another primary key within the same table.

An index is used to store a primary key so that it remains unique and ensure that a foreign key can reference it.

We used the `[Key]` attribute in the examples earlier in this chapter to define a primary key on a model.

Foreign key

A foreign key in a table is the column that is set aside to reference a primary key in a different table. Just like a primary key, a foreign key may also be a combination of multiple columns.

When you have two different tables or entities that are related to each other, there are several ways they can be related. For example, you can have one-to-one , one-to-many , and many-to-many relationships.

We mainly used a `[ForeignKey]` attribute in the examples in this chapter in order to decorate a field that we wanted to use as a foreign key, but we can also use the Fluent API to define our foreign keys. We can use the `.HasForeignKey()` property for this:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity(typeof(GameSessionModel))
      .HasOne(typeof(UserModel), "User2")
      .WithMany()
      .HasForeignKey("User2Id")
      .OnDelete(DeleteBehavior.Restrict);
    }
```

The preceding code snippet has several other attributes that we can use to define relationships between tables, examples of which include `.HasOne()` and `.WithMany()`. We will explain these later in this chapter.

WOW! eBook
www.wowebook.org

One-to-one relationships

When we have a one-to-one relationship, we can say that a record in one table will only be able to have a relationship with exactly one record in another table.

For example, if we have a `User` class and a `UserAvatar` class, and a user has one and only one avatar, then we can represent this using the Fluent API, as follows:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .HasOne(u => u.UserAvatar)
        .WithOne(a => a.User)
        .HasForeignKey<UserAvatar>(u => u.UserForeignKey);
}
```

In the preceding code snippet, we have a one-to-one relationship between the user and their avatar. This is represented by the code in bold, that is, `.HasOne(u => u.UserAvatar).WithOne(a => a.User)`. It applies both ways.

One-to-many relationships

When we have a one-to-many relationship, we can say that a record in one table is related to multiple records in another table.

An example of a one-to many relationship would be between a user and game sessions. Assuming that we have a `User` class and a `GameSession` class, the one-to-many relationship can be represented with the Fluent API as follows:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
            .HasOne(u => u.GameSession)
            .WithMany(g => g.User)
            .HasForeignKey<GameSession>(u => u.UserForeignKey);
    }
```

The preceding code snippet still uses the `.HasOne()` property, but the difference is that it is chained to a `.HasMany()` property, making it a one-to many relationship.

WOW! eBook
www.wowebook.org

Many-to-many relationships

When we have a many-to-many relationship, we can say that a record in one table is related to multiple records in another table in a specified relationship, and the other way round is also true.

An example is a relationship between students and courses.

With Entity Framework Core 3.0, you can't have the Fluent API create a many-to-many relationship directly, but there is a workaround you can use. To do this, we can construct a join table and map the two classes. Let's take a look at how to do this.

Let's say we have a class called `Student` and a class called `Course`. With these classes, we can create a `StudentCourse` table, as follows:

```
public class Student
{
   public long Id { get; set; }
   public string Name { get; set; }
   public StudentDetails StudentDetails { get; set; }
   public ICollection<StudentSubject> StudentSubjects { get; set; }
     // Added after constructed table
}
```

Then, we can have a `Course` table and a `StudentCourse` join table, as follows:

```
public class Course
{
    public long Id { get; set; }
    public string CourseName { get; set; }

    public ICollection<StudentCourse> StudentCourses { get; set; }
    // Added after constructed table
}

public class StudentCourse
{
    public long StudentId { get; set; }
    public Student Student { get; set; }

    public long CourseId { get; set; }
    public Course Course { get; set; }
}
```


WOW! eBook
www.wowebook.org

Now, we can use the Fluent API to represent our many-to-many relationship, as follows:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasKey(s => new { s.StudentId, s.SubjectId });

        modelBuilder.HasOne(ss => ss.Student)
            .WithMany(s => s.StudentSubjects)
            .HasForeignKey(ss => ss.StudentId);

        modelBuilder.HasOne(ss => ss.Subject)
            .WithMany(s => s.StudentSubjects)
            .HasForeignKey(ss => ss.SubjectId);
    }
```

Here, we have made a workaround so that we can have a many-to-many relationship between the student and course entities.

With these relationships, Entity Framework will create the necessary tables when we perform migrations and update the database, but what good is data that just sits in the database? We need to be able to query and utilize it. In the next section, we will talk about working with queries.

Working with queries

There are several ways you can retrieve data that you have saved in a database, including raw SQL statements, but by far the most convenient and safest choice is to use LINQ.

In this section, we will go through a couple of the most typical examples of querying a database using LINQ.

Querying for one item

If we had to get a game session, we would use the following code:

```
using (var context = new GameDbContext())
{
    var gameSession = context.GameSessions
        .SingleOrDefault
            (g => g.GameSessionId == Guid.Parse("002e6431-3eb5-
             4d98-b3d9-3263490ce7c0"));
}
```


WOW! eBook
www.wowebook.org

Querying for all items

If we had to return all the game sessions we've played so far, we would use the following code:

```
using (var context = new GameDbContext())
{
    var gameSessions = context.GameSessions.ToList();
}
```

Querying for filtered items

Let's say we have the following code:

```
using System.Linq;

using (var db = new GameDbContext())
{
   var users = db.Users
         .Where(u => u.GamesPlayed > 5)
         .OrderBy(u => u.FirstName)
         .ToList();
}
```

In the preceding code, we are trying to hypothetically return all the users who have played more than `5` games in total using LINQ. You can also use any other methods and properties that are available in LINQ, including `GroupBy`, `OrderByDescending`, and others. LINQ is a very powerful library on its own, and it is recommended that you should be familiar with it. If you are a total beginner to LINQ, it may help you to go through the basics here: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/basic-linq-query-operations`.

You will find that a tool called LINQPad, especially version 6, which can be found at `https://www.linqpad.net/LINQPad6.aspx`, is a great resource for working with LINQ in .NET Core 3.

In more complex scenarios, you may have multiple queries all operating together for one purpose, and in these situations, it may be desirable that, if one query out of the group fails, then all the others should be rolled back. This is when transactions come into play. We will look at transactions in the next section.

WOW! eBook
www.wowebook.org

Using transactions

A good database implementation will have the **Atomicity, Consistency, Isolation, and Durability** (**ACID**) properties.

Transactions are vital for maintaining data integrity in a database. They help ensure that data that's logically grouped together is treated as one item in a unit of operation. This goes a long way to ensure ACID properties are preserved.

When a unit of operation is triggered and saved to the database, all of the constituent logically grouped operations are successfully saved. This is what is called a **transaction**. If part of the transaction fails, then everything is rolled back. Transactions operate in an all-or-nothing scenario.

Microsoft SQL Server database is one of the many databases that actually support transactions. This means that, if we call the `SaveChanges()` method using Entity Framework Core, then every change gets treated as part of a transaction, so all the changes get saved – or none get saved in the case of an error.

You don't necessarily need to implement transactions in your own custom way, especially in most basic applications that you may be required to develop.

For most applications, this default behavior is sufficient. You should only manually control transactions if your application requirements deem it necessary. But if you are required to, you can place your processes into a transaction as follows:

```
using (var gameContext = new GameDbContext())
{
    using (var gameTransaction =
        gameContext.Database.BeginTransaction())
    {
        try
        {
            gameContext.GameInvitation.Add(new GameInvitation { ... });
            gameContext.SaveChanges();

            gameContext.GameSession.Add(new GameSession { ... });
            gameContext.SaveChanges();
            gameTransaction.Commit(); // Both the above
             operations will be in this transaction
        }
        catch (Exception Ex)
        {
            Console.WriteLine(Ex.Message)
        }
```


WOW! eBook
www.wowebook.org

```
        }
    }
```

In the preceding code snippet, we instantiated a new game database context and started a transaction on it with two processes that added a game invitation and a game session. These are committed in one go, and if any of them fail individually, then none of them will be saved at all – it's all or nothing.

This is a good point to end our discussion of Entity Framework Core 3.0 for the purposes of this book, which mainly focuses on ASP.NET Core 3.

Summary

In this chapter, we have learned how to use Entity Framework Core 3 with ASP.NET Core 3 in order to work with SQL Server databases.

We have seen how to use a database context and connection string to connect to a SQL Server database. Then, we updated the models in the Tic-Tac-Toe application with primary and foreign key definitions by using Entity Framework Core 3 Data Annotations, and by overriding the `OnModelCreating` method within the database context.

We worked with Entity Framework Core 3 migrations to constantly keep the models in our code consistent with their corresponding database representations.

Furthermore, we learned how to insert, update, and query data in an easy, productive, and efficient way. We have also learned how to query databases using the Fluent API and how to use transactions.

In the next chapter, we will talk about how to secure access to our ASP.NET Core 3 applications using ASP.NET Core 3's integrated authorization features.

WOW! eBook
www.wowebook.org

10
Securing ASP.NET Core 3 Applications

In today's world of increasing digital crime and internet fraud, all modern web applications require the implementation of strong security mechanisms for preventing attacks and user identity usurpation.

Until now, we have mainly concentrated on understanding how to build efficient ASP.NET Core 3 web applications, without thinking about user authentication, authorization, or any data protection at all, but since the Tic-Tac-Toe application is getting more and more complicated, we will have to address security issues before finally deploying it to the public.

Building a web application and not thinking about security would be a big fail and could bring down even the greatest and most famous websites. In the case of security breaches and personal data theft, the negative reputation and user confidence impacts could be tremendous, and nobody would want to work with those applications and—more troublesome—companies anymore.

This is a topic that needs to be taken very seriously. You should work with security companies to execute code verifications and intrusion tests to ensure that you comply with best practices and high-security standards (the OWASP Top 10, for example, can be found here: `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`).

Luckily, ASP.NET Core 3 contains everything necessary to help you with this complicated, but important, topic. Most of the built-in features do not even require advanced programming or security skills. You will see that it is very easy to understand and implement secure applications by using the ASP.NET Core 3 Identity framework.

The main skills that you will learn in this chapter include how to authenticate users for your application and how to authorize your users to be able to carry out different tasks in the application. You will learn how to use different types of authentication, including how to implement two-factor authentication.

WOW! eBook
www.wowebook.org

We will naturally start by looking at implementing authentication, and then implementing authorization. In authentication, we will first look at basic forms authentication, before then looking at adding external authentication, working with two-factor authentication, and finishing up by adding mechanisms for forgotten passwords and resetting mechanisms, before we then tackle authorization as a whole.

In this chapter, we will cover the following topics:

- Implementing authentication:
    - Adding basic user form authentication
    - Adding external provider authentication
    - Adding forgotten password and password reset mechanisms
    - Working with two-factor authentication
- Implementing authorization

Implementing authentication

Authentication allows applications to *identify* a specific user. It is not used to manage user access rights, which is the role of authorization, nor is it used to protect data, which is the role of data protection.

There are several methods for authenticating application users, such as the following:

- Basic user form authentication, using a login form with login and password boxes
- **Single Sign-On** (**SSO**) authentication, where the user only authenticates once for all their applications within the context of their company
- Social network external provider authentication (such as Facebook and LinkedIn)
- Certificate or **Public Key Infrastructure** (**PKI**) authentication

ASP.NET Core 3 supports all these methods, but in this chapter, we will concentrate on forms authentication with a user login and password, and external provider authentication via Facebook.

In the following examples, you will see how to use those methods to authenticate application users, along with a number of more advanced features, such as email confirmation and password reset mechanisms.

And last but not the least, you will see how to implement two-factor authentication using the built-in ASP.NET Core 3 authentication features for your most critical applications.

WOW! eBook
www.wowebook.org

Let's prepare the implementation of the different authentication mechanisms for the Tic-Tac-Toe application:

1. Update the lifetime of `UserService`, `GameInvitationService`, and `GameSessionService` in the `Startup` class:

```
services.AddTransient<IUserService, UserService>();
services.AddScoped<IGameInvitationService,
 GameInvitationService>();
services.AddScoped<IGameSessionService, GameSessionService>
();
```

2. Update the `Configure` method within the `Startup` class, and call the authentication middleware directly after the Static Files Middleware:

```
app.UseStaticFiles();
app.UseAuthentication();
```

3. Update `UserModel` to use it with the built-in ASP.NET Core Identity authentication features, and remove the `Id` and `Email` properties, which are already provided by the `IdentityUser` class:

```
public class UserModel : IdentityUser<Guid>
{
  [Display(Name = "FirstName")]
  [Required(ErrorMessage = "FirstNameRequired")]
  public string FirstName { get; set; }
  [Display(Name = "LastName")]
  [Required(ErrorMessage = "LastNameRequired")]
  public string LastName { get; set; }
  [Display(Name = "Password")]
  [Required(ErrorMessage = "PasswordRequired"),
    DataType(DataType.Password)]
  public string Password { get; set; }
  [NotMapped]
  public bool IsEmailConfirmed{ get {
     return EmailConfirmed; }  }
  public System.DateTime? EmailConfirmationDate { get; set;
}
  public int Score { get; set; }
}
```

Note that in the real world, we would advise also removing the `Password` property. However, we will keep it in the example for clarity and learning purposes.

WOW! eBook
www.wowebook.org

4. Add a new folder called `Managers`, add a new manager in the folder called `ApplicationUserManager`, and then add the following constructor:

```
public class ApplicationUserManager : UserManager<UserModel>
{
private IUserStore<UserModel> _store;
DbContextOptions<GameDbContext> _dbContextOptions;
public ApplicationUserManager(DbContextOptions<GameDbContext>
dbContextOptions,
IUserStore<UserModel> store, IOptions<IdentityOptions>
optionsAccessor, IPasswordHasher<UserModel> passwordHasher,
IEnumerable<IUserValidator<UserModel>>
userValidators,IEnumerable<IPasswordValidator<UserModel>>
passwordValidators, ILookupNormalizer
Normalizer,IdentityErrorDescriber errors, IServiceProvider
services,
ILogger<UserManager<UserModel>> logger) :
      base(store, optionsAccessor, passwordHasher, userValidators,
       passwordValidators, keyNormalizer, errors, services, logger)
        {
            _store = store;
            _dbContextOptions = dbContextOptions;
        }
          ...
 }
```

Let's have a look at the steps to have a fully functioning `ApplicationUserManager` class:

1. Add a `FindByEmailAsync` method as follows:

```
public override async Task<UserModel> FindByEmailAsync
  (string email)
{
  using (var dbContext = new GameDbContext
    (_dbContextOptions))
  {
    return await dbContext.Set<UserModel>
     ().FirstOrDefaultAsync(
     x => x.Email == email);
  }
}
```

2. Add a `FindByIdAsync` method as follows:

```
public override async Task<UserModel> FindByIdAsync(string
  userId)
```


WOW! eBook
www.wowebook.org

```
        {
          using (var dbContext = new GameDbContext
           (_dbContextOptions))
          {
            Guid id = Guid.Parse(userId);
            return await dbContext.Set<UserModel>
             ().FirstOrDefaultAsync(
             x => x.Id == id);
          }
        }
```

3. Add an `UpdateAsync` method as follows:

```
public override async Task<IdentityResult> UpdateAsync
    (UserModel user)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
   var current = await dbContext.Set<UserModel>
    ().FirstOrDefaultAsync(x => x.Id == user.Id);
   current.AccessFailedCount = user.AccessFailedCount;
   current.ConcurrencyStamp = user.ConcurrencyStamp;
   current.Email = user.Email;
   current.EmailConfirmationDate = user.EmailConfirmationDate;
   current.EmailConfirmed = user.EmailConfirmed;
   current.FirstName = user.FirstName;
   current.LastName = user.LastName;
   current.LockoutEnabled = user.LockoutEnabled;
   current.NormalizedEmail = user.NormalizedEmail;
   current.NormalizedUserName = user.NormalizedUserName;
   current.PhoneNumber = user.PhoneNumber;
   current.PhoneNumberConfirmed = user.PhoneNumberConfirmed;
   current.Score = user.Score;
   current.SecurityStamp = user.SecurityStamp;
   current.TwoFactorEnabled = user.TwoFactorEnabled;
   current.UserName = user.UserName;
   await dbContext.SaveChangesAsync();
   return IdentityResult.Success;
   }
  }
```

4. Add a `ConfirmEmailAsync` method as follows:

```
public override async Task<IdentityResult>
ConfirmEmailAsync(UserModel user, string token)
  {
    var isValid = await base.VerifyUserTokenAsync(user,
     Options.Tokens.EmailConfirmationTokenProvider,
```


WOW! eBook
www.wowebook.org

```
                ConfirmEmailToken
                Purpose, token);
               if (isValid)
               {
                 using (var dbContext =  new GameDbContext
                   (_dbContextOptions))
                 {
                   var current =  await dbContext.UserModels.
                    FindAsync(user.Id);
                   current.EmailConfirmationDate = DateTime.Now;
                   current.EmailConfirmed = true;
                   await dbContext.SaveChangesAsync();
                   return IdentityResult.Success;
                  }
                }
                return IdentityResult.Failed();
                }
              }
```

5. Update the `Startup` class, and register the `ApplicationUserManager` class:

```
services.AddTransient<ApplicationUserManager>();
```

6. Update `UserService` to work with the `ApplicationUserManager` class, with the constructor as follows:

```
public class UserService : IUserService
{
  private ILogger<UserService> _logger;
  private ApplicationUserManager _userManager;
  public UserService(ApplicationUserManager userManager,
   ILogger<UserService> logger)
  {
    _userManager = userManager;
    _logger = logger;

    var emailTokenProvider = new EmailTokenProvider<UserModel>();
    _userManager.RegisterTokenProvider("Default",
      emailTokenProvider);
  }
  ...
}
```


WOW! eBook
www.wowebook.org

The following additions are done to make use of the `ApplicationUserManager` class, register the authentication middleware and then prepare the database:

1. Add two new methods, the first one called `GetEmailConfirmationCode`, as follows:

```
public async Task<string> GetEmailConfirmationCode
  (UserModel user)
{
  return await _userManager.
   GenerateEmailConfirmationTokenAsync(user);
}
```

2. Secondly, add a `ConfirmEmail` method as follows:

```
public async Task<bool> ConfirmEmail(string email, string code)
{
  var start = DateTime.Now;
  _logger.LogTrace($"Confirm email for user {email}");

  var stopwatch = new Stopwatch(); stopwatch.Start();

   try
   {
      var user = await _userManager.FindByEmailAsync(email);
      if (user == null) return false;
      var result = await _userManager.ConfirmEmailAsync(user,
        code);
      return result.Succeeded;
    }
    catch (Exception ex)
    {
       _logger.LogError($"Cannot confirm email for user
         {email} - {ex}");
       return false;
    }
    finally
    {
       stopwatch.Stop();
       _logger.LogTrace($"Confirm email for user finished in
        {stopwatch.Elapsed}");
    }
 }
```


WOW! eBook
www.wowebook.org

3. Update the `RegisterUser` method as follows:

```
public async Task<bool> RegisterUser(UserModel userModel)
{
    var start = DateTime.Now;
    _logger.LogTrace($"Start register user {userModel.Email} -
     {start}");

    var stopwatch = new Stopwatch();  stopwatch.Start();

    try
    {
      userModel.UserName = userModel.Email;
      var result = await _userManager.CreateAsync
        (userModel,userModel.Password);
      return result == IdentityResult.Success;
     }
     catch (Exception ex)
     {
       _logger.LogError($"Cannot register user
        {userModel.Email} -
        {ex}");
       return false;
     }
     finally
     {
       stopwatch.Stop();
       _logger.LogTrace($"Start register user {userModel.Email}
         finished at {DateTime.Now} - elapsed
         {stopwatch.Elapsed.
         TotalSeconds} second(s)");
      }
   }
```

4. Update the `GetUserByEmail`, `IsUserExisting`, `GetTopUsers`, and `UpdateUser` methods and then update the user service interface:

```
     public async Task<UserModel> GetUserByEmail(string
       email)
    {
      return await _userManager.FindByEmailAsync(email);
    }

    public async Task<bool> IsUserExisting(string email)
    {
      return (await _userManager.FindByEmailAsync(email)) !=
       null;
```


WOW! eBook
www.wowebook.org

```
  }

  public async Task<IEnumerable<UserModel>> GetTopUsers(
    int numberOfUsers)
  {
    return await _userManager.Users.OrderByDescending( x =>
     x.Score).ToListAsync();
  }

  public async Task UpdateUser(UserModel userModel)
  {
    await _userManager.UpdateAsync(userModel);
  }
```

Note that you should also update the `UserServiceTest` class to work with the new constructor. For that, you will also have to create a mock for the `UserManager` class and pass it to the constructor. For the moment, you can just disable the unit test by commenting it out and updating it later. But don't forget to do it!

7. Update the `EmailConfirmation` method in `UserRegistrationController`, and use the `GetEmailConfirmationCode` method you have added previously to retrieve the email code:

```
var urlAction = new UrlActionContext
{
  Action = "ConfirmEmail",
  Controller = "UserRegistration",
  Values = new { email, code =
   await _userService.GetEmailConfirmationCode(user) },
  Protocol = Request.Scheme,
  Host = Request.Host.ToString()
};
```

8. Update the `ConfirmEmail` method in `UserRegistrationController`; it has to call the `ConfirmEmail` method in `UserService` to finish the email confirmation:

```
[HttpGet]
public async Task<IActionResult> ConfirmEmail(string email,
 string code)
{
  var confirmed = await _userService.ConfirmEmail(email,
    code);

  if (!confirmed)
    return BadRequest();
```


WOW! eBook
www.wowebook.org

```
    return RedirectToAction("Index", "Home");
  }
```

9. Add a new class called `RoleModel` in the `Models` folder, and make it inherit from `IdentityRole<long>`, as it will be used by the built-in ASP.NET Core Identity Authentication features:

```
public class RoleModel : IdentityRole<Guid>
{
  public RoleModel()
  {
  }

  public RoleModel(string roleName) : base(roleName)
  {
  }
}
```

10. Update `GameDbContext`, and add a new `DbSet` for role models:

```
public DbSet<RoleModel> RoleModels { get; set; }
```

11. Register the authentication service and the identity service in the `Startup` class, and then use the new role model you added previously:

```
services.AddIdentity<UserModel, RoleModel>(options =>
{
  options.Password.RequiredLength = 1;
  options.Password.RequiredUniqueChars = 0;
  options.Password.RequireNonAlphanumeric = false;
  options.Password.RequireUppercase = false;
  options.SignIn.RequireConfirmedEmail = false;
}).AddEntityFrameworkStores<GameDbContext>
  ().AddDefaultTokenProviders();

services.AddAuthentication(options => {
  options.DefaultScheme = CookieAuthenticationDefaults.
    AuthenticationScheme;
  options.DefaultSignInScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
}).AddCookie();
```


WOW! eBook
www.wowebook.org

12. Update the communication middleware, remove the `_userService` private member from the class, and update the constructor accordingly:

```
public CommunicationMiddleware(RequestDelegate next)
{
  _next = next;
}
```

13. Update the two `ProcessEmailConfirmation` methods in the communication middleware, as they must be asynchronous in order to work with ASP.NET Core Identity. Stop using the privately defined `private readonly IUserService _userService;` user service, in preference to a locally defined user service in each of the two methods as follows:

```
private async Task ProcessEmailConfirmation(HttpContext
context,
    WebSocket currentSocket, CancellationToken ct, string
    email)
{
  var userService = context.RequestServices.
    GetRequiredService<IUserService>();
  ...
}

private async Task ProcessEmailConfirmation(HttpContext
 context)
{
  var userService =  context.RequestServices.
    GetRequiredService<IUserService>();
  ...
}
```

14. Update `GameInvitationService`, and set the public constructor to `static`.
15. Remove the following `DbContextOptions` registration from the `Startup` class; this will be replaced by another one in the next step:

```
var dbContextOptionsbuilder =
  new DbContextOptionsBuilder<GameDbContext>()
   .UseSqlServer(connectionString);
services.AddSingleton(dbContextOptionsbuilder.Options);
```

16. Update the `Startup` class, and add a new `DbContextOptions` registration:

```
var connectionString = Configuration.
GetConnectionString("DefaultConnection");
services.AddScoped(typeof(DbContextOptions<GameDbContext>),
        (serviceProvider) =>
```


WOW! eBook
www.wowebook.org

```
{
  return new DbContextOptionsBuilder<GameDbContext>()
   .UseSqlServer(connectionString).Options;
});
```

17. Update the `Configure` method in the `Startup` class, and then replace the code that executes the database migration at the end of the method:

```
var provider = app.ApplicationServices;
var scopeFactory = provider.
 GetRequiredService<IServiceScopeFactory>();
using (var scope = scopeFactory.CreateScope())
using (var context = scope.ServiceProvider.
 GetRequiredService<GameDbContext>())
{
  context.Database.Migrate();
}
```

18. Update the `Index` method in `GameInvitationController`:

```
...
var invitation =
  gameInvitationService.Add(gameInvitationModel).Result;
return RedirectToAction("GameInvitationConfirmation",
 new { id = invitation.Id });
...
```

19. Update the `ConfirmGameInvitation` method in `GameInvitationController`, and add additional fields to the existing user registration:

```
await _userService.RegisterUser(new UserModel
{
  Email = gameInvitation.EmailTo,
  EmailConfirmationDate = DateTime.Now,
  EmailConfirmed = true,
  FirstName = "",
  LastName = "",
  Password = "Qwerty123!",
  UserName = gameInvitation.EmailTo
});
```

Note that the automatic creation and registration of the invited user is only a temporary workaround that we have added to simplify the example application. In the real world, you will need to handle this case differently and replace the temporary workaround with a real solution.

WOW! eBook
www.wowebook.org

20. Update the `CreateGameSession` method in `GameSessionService` by passing in the `invitedBy` and `invitedPlayer` user models, instead of defining them internally as previously:

```
public async Task<GameSessionModel> CreateGameSession(
         Guid invitationId, UserModel invitedBy, UserModel
           invitedPlayer)
        {
          var session = new GameSessionModel
          {
            User1 = invitedBy,
            User2 = invitedPlayer,
            Id = invitationId,
            ActiveUser = invitedBy
          };
          _sessions.Add(session);
          return session;
        }
```

Update the `AddTurn` method in `GameSessionService` by passing in a user instead of getting the user by email as before, and then re-extract the `GameSessionService` interface:

```
public async Task<GameSessionModel> AddTurn(Guid id, UserModel
    user, int x, int y)
    {
      ...
      turns.Add(new TurnModel
        {
            User = user,
            X = x,
            Y = y,
            IconNumber = user.Email == gameSession.User1?
            .Email ? "1" : "2"
        });

        gameSession.Turns = turns;
        gameSession.TurnNumber = gameSession.TurnNumber + 1;

        if (gameSession.User1?.Email == user.Email)
            gameSession.ActiveUser = gameSession.User2;
      ...
    }
```


WOW! eBook
www.wowebook.org

21. Update the `Index` method in `GameSessionController`:

```
public async Task<IActionResult> Index(Guid id)
{
  var session = await _gameSessionService.GetGameSession(id);
  var userService = HttpContext.RequestServices.
    GetService<IUserService>();
  if (session == null)
  {
 var gameInvitationService = quest.HttpContext.RequestServices.
    GetService<IGameInvitationService>();
    var invitation = await gameInvitationService.Get(id);
    var invitedPlayer = await userService.GetUserByEmail
     (invitation.EmailTo);
    var invitedBy = await userService.GetUserByEmail
     (invitation.InvitedBy);
    session = await _gameSessionService.CreateGameSession(
     invitation.Id, invitedBy, invitedPlayer);
    }
    return View(session);
}
```

22. Update the `SetPosition` method in `GameSessionController`, and pass `turn.User` instead of `turn.User.Email` (make sure that `IGameSessionService` has the following definition: `Task<GameSessionModel> AddTurn(Guid id, UserModel user, int x, int y);`):

```
gameSession = await _gameSessionService.AddTurn(gameSession.Id,
 turn.User, turn.X, turn.Y);
```

23. Update the `OnModelCreating` method in `GameDbContext`, and add a `WinnerId` foreign key:

```
...
modelBuilder.Entity(typeof(GameSessionModel))
 .HasOne(typeof(UserModel), "Winner")
 .WithMany()
 .HasForeignKey("WinnerId")
 .OnDelete(DeleteBehavior.Restrict);
...
```


WOW! eBook
www.wowebook.org

24. Update the `GameInvitationConfirmation` method in `GameInvitationController` to make it asynchronous. A controller action must be asynchronous in order to work with ASP.NET Core Identity:

```
[HttpGet]
public async Task<IActionResult>
GameInvitationConfirmation(
 Guid id, [FromServices]IGameInvitationService
 gameInvitationService)
{
  return await Task.Run(() =>
  {
    var gameInvitation = gameInvitationService.Get(id).
     Result;
    return View(gameInvitation);
  });
}
```

25. Update the `Index` and `SetCulture` methods in `HomeController` so that they are asynchronous in order to work with ASP.NET Core Identity:

```
public async Task<IActionResult> Index()
{
  return await Task.Run(() =>
  {
    var culture = Request.HttpContext.Session.
    GetString("culture");
    ViewBag.Language = culture; return View();
  });
}

public async Task<IActionResult> SetCulture(string culture)
{
  return await Task.Run(() =>
  {
    Request.HttpContext.Session.SetString("culture",
     culture);
    return RedirectToAction("Index");
  });
}
```

26. Update the `Index` method in `UserRegistrationController` and make it asynchronous to work with ASP.NET Core Identity:

```
public async Task<IActionResult> Index()
{
  return await Task.Run(() =>
```


WOW! eBook
www.wowebook.org

```
        {
          return View();
        });
      }
```

27. Open the Package Manager Console and execute the `Add-Migration IdentityDb` command.
28. Update the database by executing the `Update-Database` command in the Package Manager Console.
29. Start the application and register a new user, and then verify that everything is still working as expected.

Note that you have to use a complex password, such as `Azerty123!`, to be able to finish the user registration successfully now, since you have implemented the integrated features of ASP.NET Core Identity in this section, which requires complex passwords.

Well done for reaching this far, as our application is now ready to use ASP.NET Core Identity and, in general, it is now ready to handle different types of authentication, after all the preparation work in the preceding section. We are now at a good place to start learning how to add different types of authentication, and we start in the next section by looking at basic user form authentication.

Adding basic user form authentication

Great! You have registered the authentication middleware and prepared the database. In the next step, you are going to implement basic user authentication for the Tic-Tac-Toe application.

The following example demonstrates how to modify the user registration and add a simple login form with a user login and password textbox for authenticating users:

1. Add a new model called `LoginModel` to the `Models` folder:

```
public class LoginModel
{
  [Required]
  public string UserName { get; set; }
  [Required]
  public string Password { get; set; }
  public string ReturnUrl { get; set; }
}
```


WOW! eBook
www.wowebook.org

2. Add a new folder called `Account` to the `Views` folder, and then add a new file called `Login.cshtml` within this new folder. It will contain the login view:

```
@model TicTacToe.Models.LoginModel
<div class="container">
  <div id="loginbox" style="margin-top:50px;"
   class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Sign In</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div style="display:none" id="login-alert"
         class="alert alert-danger col-sm-12"></div>
        <form id="loginform" class="form-horizontal"
         role="form" asp-action="Login" asp-
          controller="Account">
          <input type="hidden" asp-for="ReturnUrl" />
          <div asp-validation-summary="ModelOnly"
           class="text-danger"></div>
          <div style="margin-bottom: 25px" class="input-
            group">
            <span class="input-group-addon"><i
             class="glyphicon
             glyphicon-user"></i></span>
            <input type="text" class="form-control"
             asp-for="UserName" value=""
              placeholder="username
             or email">
          </div>
          <div style="margin-bottom: 25px" class="input-
            group">
            <span class="input-group-addon"><i
              class="glyphicon
             glyphicon-lock"></i></span>
            <input type="password" class="form-control"
             asp-for="Password" placeholder="password">
          </div>
          <div style="margin-top:10px" class="form-group">
            <div class="col-sm-12 controls">
              <button type="submit" id="btn-login" href="#"
               class="btn btn-success">Login</button>
            </div>
          </div>
          <div class="form-group">
            <div class="col-md-12 control">
              <div style="border-top: 1px solid#888;
```


WOW! eBook
www.wowebook.org

```
                                padding-top:15px; font-size:85%">
                                 Don't have an account?
                                 <a asp-action="Index"
                                  asp-controller="UserRegistration">Sign Up
                                   Here
                                 </a>
                               </div>
                             </div>
                           </div>
                         </form>
                       </div>
                     </div>
                   </div>
                 </div>
```

3. Update `UserService`, add a `SignInManager` private field, and then update the constructor:

```
...
private SignInManager<UserModel> _signInManager;
public UserService(ApplicationUserManager userManager,
 ILogger<UserService> logger, SignInManager<UserModel>
 signInManager)
{
  ...
  _signInManager = signInManager;
  ...
}
...
```

4. Add a new method called `SignInUser` to `UserService`:

```
public async Task<SignInResult> SignInUser( LoginModel loginModel,
HttpContext httpContext)
{
    _logger.LogTrace($"signin user {loginModel.UserName}");

    var stopwatch = new Stopwatch(); stopwatch.Start();
    try
     {
       var user = await _userManager.FindByNameAsync
            (loginModel.UserName);
       var isValid = await _signInManager.CheckPasswordSignInAsync
        (user, loginModel.Password, true);
       if (!isValid.Succeeded)  return SignInResult.Failed;
       if (!await _userManager.IsEmailConfirmedAsync(user))
           return SignInResult.NotAllowed;
```


WOW! eBook
www.wowebook.org

```
            var identity = new ClaimsIdentity
             (CookieAuthenticationDefaults.AuthenticationScheme);
                    identity.AddClaim(new Claim(ClaimTypes.Name,
                     loginModel.UserName));
                    identity.AddClaim(new Claim(ClaimTypes.GivenName,
                     user.FirstName));
                    identity.AddClaim(new Claim(ClaimTypes.Surname,
                     user.LastName));
                    identity.AddClaim(new Claim("displayName", $"
                     {user.FirstName} {user.LastName}"));

            if (!string.IsNullOrEmpty(user.PhoneNumber))
                identity.AddClaim(new Claim(ClaimTypes.HomePhone,
                 user.PhoneNumber));
            identity.AddClaim(new Claim("Score", user.Score.
             ToString()));

            await httpContext.SignInAsync(CookieAuthenticationDefaults.
             AuthenticationScheme,
           new ClaimsPrincipal(identity), new AuthenticationProperties {
             IsPersistent = false });

           return isValid;
         }
         catch (Exception ex)
         {
          _logger.LogError($"cannot sign in user{ loginModel.UserName} -
{
            ex} ");
          throw ex;
         }
         finally
         {
           stopwatch.Stop();
         _logger.LogTrace($"sign in user {loginModel.UserName} finished
           in
           { stopwatch.Elapsed} ");
         }
       }
```

Add another method, `SignOutUser`, to `UserService` and update the user service interface:

```
public async Task SignOutUser(HttpContext httpContext)
        {
          await _signInManager.SignOutAsync();
          await httpContext.SignOutAsync(new
           AuthenticationProperties {
```


WOW! eBook
www.wowebook.org

```
                IsPersistent = false });
              return;
            }
```

5. Add a new controller called `AccountController` to the `Controllers` folder:

```
public class AccountController : Controller
        {
          private IUserService _userService;
          public AccountController(IUserService userService)
          {
            _userService = userService;
          }
         }
```

Let's perform the steps as follows:

1. Implement a new `Login` method in `AccountController` as follows:

```
public async Task<IActionResult> Login(string returnUrl)
 {
   return await Task.Run(() =>
   {
     var loginModel = new LoginModel { ReturnUrl =
      returnUrl };
     return View(loginModel);
   });
 }
...
```

2. Add another implementation of the `Login` method that takes in a login model as a parameter:

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel
loginModel)
{
  if (ModelState.IsValid)
   {
    var result = await _userService.SignInUser(loginModel,
     HttpContext);

    if (result.Succeeded)
    {
      if (!string.IsNullOrEmpty(loginModel.ReturnUrl))
          return Redirect(loginModel.ReturnUrl);
      else  return RedirectToAction("Index", "Home");
```


WOW! eBook
www.wowebook.org

```
    }
    else     ModelState.AddModelError("",
result.IsLockedOut ?"User
      is locked" : "User is not allowed");
  }
  return View();
}
```

3. Add a new `Logout` method as follows:

```
public IActionResult Logout()
{
  _userService.SignOutUser(HttpContext).Wait();
  HttpContext.Session.Clear();
  return RedirectToAction("Index", "Home");
}
```

6. Update the `Views/Shared/_Menu.cshtml` file, and replace the existing code block at the top of the method:

```
@using Microsoft.AspNetCore.Http;
@{
  var email = User?.Identity?.Name ??
    Context.Session.GetString("email");
  var displayName = User.Claims.FirstOrDefault(
    x => x.Type == "displayName")?.Value ??
    Context.Session.GetString("displayName");
}
```

7. Update the `Views/Shared/_Menu.cshtml` file to display either a display name element for already authenticated users or a login element for an authenticated user; for that, replace the final `<li>` element:

```
<li>
  @if (!string.IsNullOrEmpty(email))
  {
    Html.RenderPartial("_Account",
     new TicTacToe.Models.AccountModel { Email = email,
     DisplayName = displayName });
  }
  else
  {
    <a asp-area="" asp-controller="Account"
     asp-action="Login">Login</a>
  }
</li>
```


WOW! eBook
www.wowebook.org

8. Update the `Views/Shared/_Account.cshtml` file, and replace the **Log Off** and **View Details** links:

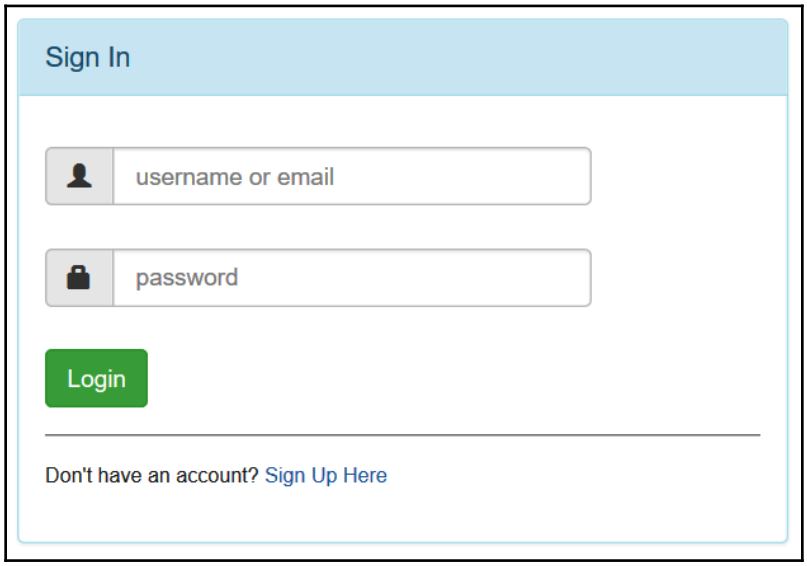
```
        <a class="btn btn-danger btn-block" asp-
controller="Account"
         asp-action="Logout" asp-area="">Log Off</a>
        <a class="btn btn-default btn-block" asp-action="Index"
         asp-controller="Home" asp-area="Account">View Details</a>
```

9. Go to the `Views\Shared\Components\GameSession` folder, and update the `default.cshtml` file to improve the visual representation by having our table as follows:

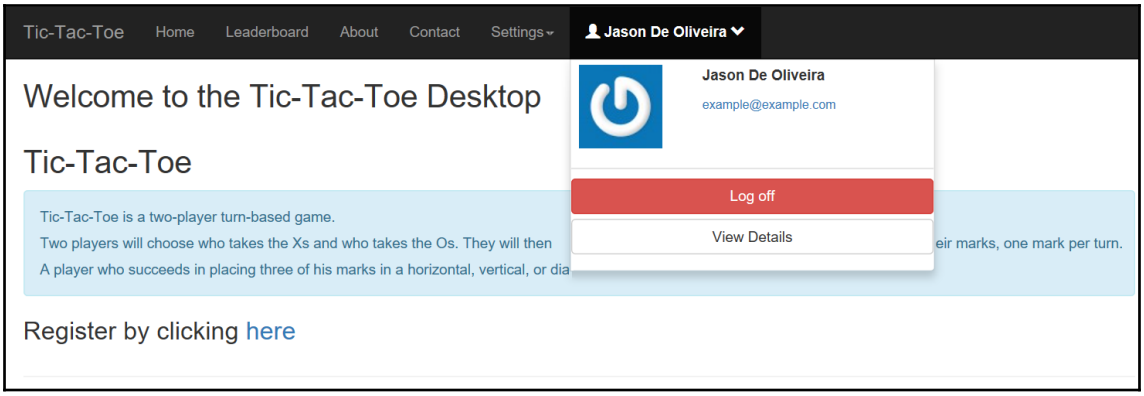
```
...
<table>
  @for (int rows = 0; rows < 3; rows++)
  {
    <tr style="height:150px;">
      @for (int columns = 0; columns < 3; columns++)
      {
        <td style="width:150px; border:1px solid #808080;text-
         align:center; vertical-align:middle"
          id="@($"c_{rows}_{columns}")">
          @{
            var position = Model.Turns?.FirstOrDefault(turn =>
             turn.X == columns && turn.Y == rows);
            if (position != null)
            {
              if (position.User == Model.User1)
                <i class="glyphicon glyphicon-unchecked"></i>
              else
                <i class="glyphicon glyphicon-remove-circle"></i>
            }
            else
            {
              <a class="btn btn-default btn-SetPosition"style=
               "width:150px; min-height:150px;"
                 data-X="@columns" data-Y="@rows">   </a>
            }
          }
        </td>
      }
    </tr>
  }
</table>
 ...
```


WOW! eBook
www.wowebook.org

10. Start the application, click on the **Login** element in the top menu, and sign in as an existing user (or register as a user if you have not done so previously):



11. Click the **Log Off** button. You should be logged off and get redirected back to the **Home** page:



That essentially makes up our forms authentication, where we have been able to sign a user in and out with a login form. In the next section, we will look at how we can add an external provider as a means of authentication to our application.

WOW! eBook
www.wowebook.org

Adding external provider authentication

In the following section, we will showcase external provider authentication by using Facebook as an authentication provider.

Here is an overview of the control flow in this case:

1. The user clicks on a dedicated external provider login button.
2. The corresponding controller receives a request indicating which provider is needed, and then a challenge is initiated with the external provider.
3. The external provider sends an HTTP callback (`POST` or `GET`) with a provider name, a key, and some user claims for the application.
4. The claims are matched with the internal application user.
5. If no internal user can be matched with the claims, the user is either redirected to a specific registration form or is rejected.

Note that the implementation steps are the same for all external providers if they support OWIN and ASP.NET Core Identity, and that you may even create your own providers and integrate them in the same way.

We are now going to implement external provider authentication via Facebook:

1. Update the login form, and add a button called **Login with Facebook** directly after the standard **Login** button:

```
<a id="btn-fblogin" asp-action="ExternalLogin"
 asp-controller="Account" asp-route-Provider="Facebook"
 class="btn btn-primary">Login with Facebook</a>
```

2. Update the `UserService` class and the user service interface, and then add two new methods called `GetExternalAuthenticationProperties` and `GetExternalLoginInfoAsync`:

```
public async Task<AuthenticationProperties>
 GetExternalAuthenticationProperties(string provider,
 string redirectUrl)
{
  return await Task.FromResult(
   _signInManager.ConfigureExternalAuthentication
    Properties(
   provider, redirectUrl));
}
```


WOW! eBook
www.wowebook.org

```
        public async Task<ExternalLoginInfo>
         GetExternalLoginInfoAsync()
        {
          return await _signInManager.GetExternalLoginInfoAsync();
        }
```

Add another new method called `ExternalLoginSignInAsync`:

```
public async Task<SignInResult> ExternalLoginSignInAsync(
         string loginProvider, string providerKey, bool
          isPersistent)
        {
          _logger.LogInformation($"Sign in user with external login
           {loginProvider} - {providerKey}");
          return await _signInManager.ExternalLoginSignInAsync(
           loginProvider, providerKey, isPersistent);
        }
```

3. Update `AccountController`, and add a method called `ExternalLogin` :

```
[AllowAnonymous]
public async Task<ActionResult> ExternalLogin(string provider,
string ReturnUrl)
{
  var redirectUrl = Url.Action(nameof(ExternalLoginCallBack),
   "Account", new { ReturnUrl = ReturnUrl }, Request.Scheme,
    Request.Host.ToString());
  var properties = await _userService.
    GetExternalAuthenticationProperties(provider, redirectUrl);
  ViewBag.ReturnUrl = redirectUrl;
  return Challenge(properties, provider);
}
```

In the same `AccountController` class, add another method called `ExternalLoginCallBack`:

```
[AllowAnonymous]
public async Task<IActionResult> ExternalLoginCallBack(string
returnUrl, string remoteError = null)
{
  if (remoteError != null)
  {
    ModelState.AddModelError(string.Empty, $"Error from external
     provider: {remoteError}");
    ViewBag.ReturnUrl = returnUrl;
    return View("Login");
  }
  var info = await _userService.GetExternalLoginInfoAsync();
```


WOW! eBook
www.wowebook.org

```
    if (info == null)
       return RedirectToAction("Login", new { ReturnUrl = returnUrl
});
    var result = await _userService.ExternalLoginSignInAsync(
          info.LoginProvider, info.ProviderKey, isPersistent: false);
    if (result.Succeeded)
    {
      if (!string.IsNullOrEmpty(returnUrl)) return
       Redirect(returnUrl);
      else  return RedirectToAction("Index", "Home");
    }
    if (result.IsLockedOut)  return View("Lockout");
    else  return View("NotFound");
    }
}
```

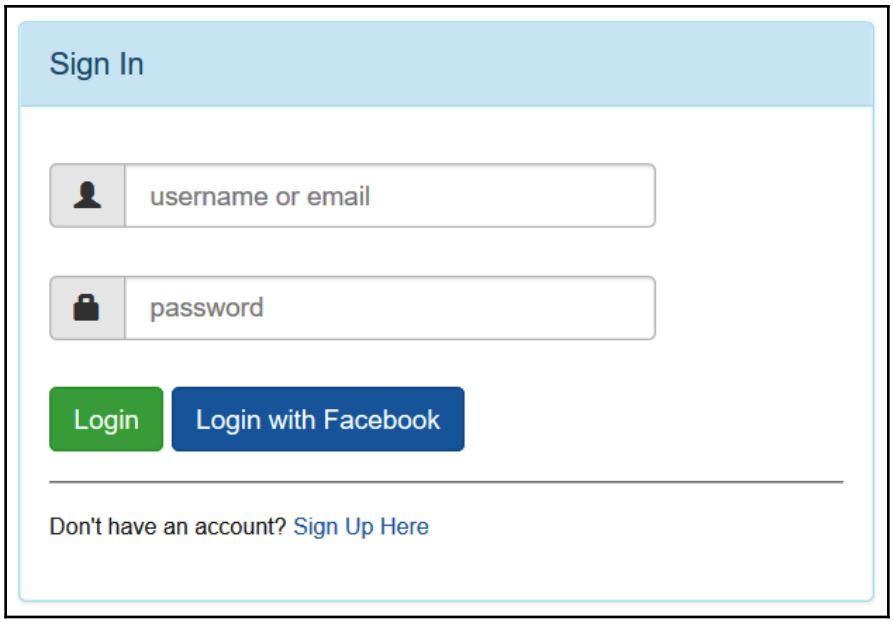
4. Register the Facebook middleware within the `Startup` class:

```
services.AddAuthentication(options => {
  options.DefaultScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultSignInScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme =
   CookieAuthenticationDefaults.AuthenticationScheme;
}).AddCookie().AddFacebook(facebook =>
{
  facebook.AppId = "123";
  facebook.AppSecret = "123";
  facebook.ClientId = "123";
  facebook.ClientSecret = "123";
});
```

Note that you must update the Facebook middleware configuration and register your application with the Facebook developer portal before being able perform authenticated logins with a Facebook account.
Please go to `https://developer.facebook.com` for more information.

5. Start the application, click on the **Login with Facebook** button, sign in with your Facebook credentials, and verify that everything is working as expected:

WOW! eBook
www.wowebook.org



Congratulations on reaching this far and, with similar steps as before, you will be able to use other external providers such as Google, or indeed Microsoft, for authentication. Now, let's look at how we can implement two-factor authentication in the next section.

Working with two-factor authentication

The standard security mechanisms you have seen before only require a simple username and password, which makes it increasingly easy for cyber criminals to gain access to confidential data, such as personal and financial details, either by hacking the password or by intercepting user credentials (emails, network sniffing, and such). This data can then be used to commit financial fraud and identity theft.

Two-factor authentication adds an extra layer of security since it requires not only a username and password, but also a two-factor code that only the user can provide (physical device, software-generated, and so on). This makes it much harder for potential intruders to gain access and thus helps to prevent identity and data theft.

WOW! eBook
www.wowebook.org

All major websites provide two-factor authentication as an option, so let's add it to the Tic-Tac-Toe application as well.

Two-factor authentication - step by step

The following steps will enable your application to have complete two-factor authentication:

1. Add a new model called `TwoFactorCodeModel` to the `Models` folder:

```
public class TwoFactorCodeModel
{
  [Key]
  public long Id { get; set; }
  public Guid UserId { get; set; }
  [ForeignKey("UserId")]
  public UserModel User { get; set; }
  public string TokenProvider { get; set; }
  public string TokenCode { get; set; }
}
```

2. Add a new model called `TwoFactorEmailModel` to the `Models` folder:

```
public class TwoFactorEmailModel
{
  public string DisplayName { get; set; }
  public string Email { get; set; }
  public string ActionUrl { get; set; }
}
```

3. Register `TwoFactorCodeModel` within `GameDbContext` by adding a corresponding `DbSet`:

```
public DbSet<TwoFactorCodeModel> TwoFactorCodeModels { get;
 set; }
```

4. Open the NuGet Package Manager Console and execute the `Add-Migration AddTwoFactorCode` command. Then, update the database by executing the `Update-Database` command.
5. Update `ApplicationUserManager`, and then add a new method called `SetTwoFactorEnabledAsync`:

```
public override async Task<IdentityResult>
SetTwoFactorEnabledAsync(UserModel user, bool enabled)
{
```


WOW! eBook
www.wowebook.org

```
    try
    {
      using (var db = new GameDbContext(_dbContextOptions))
      {
        var current = await db.UserModels.FindAsync(user.Id);
        current.TwoFactorEnabled = enabled; await
         db.SaveChangesAsync();
        return IdentityResult.Success;
      }
    }
    catch (Exception ex)
    { return IdentityResult.Failed(new IdentityError {Description =
  ex.ToString() });}
  }
```

Then we perform the following steps:

1. Add another method called `GenerateTwoFactorTokenAsync`:

```
public override async Task<string>GenerateTwoFactorTokenAsync
 (UserModel user, string tokenProvider)
{
  using (var dbContext = new GameDbContext(_dbContextOptions))
  {
    var emailTokenProvider = new EmailTokenProvider
     <UserModel>();
    var token = await emailTokenProvider.GenerateAsync
     ("TwoFactor", this, user);
    dbContext.TwoFactorCodeModels.Add(new TwoFactorCodeModel
     { TokenCode = token,TokenProvider = tokenProvider,UserId =
      user.Id });
    if (dbContext.ChangeTracker.HasChanges())
              await dbContext.SaveChangesAsync();

    return token;
  }
}
```

2. And finally, to the same `ApplicationUserManager`, add a `VerifyTwoFactorTokenAsync` method:

```
public override async Task<bool>
 VerifyTwoFactorTokenAsync(UserModel user, string
  tokenProvider, string token)
{
  using (var dbContext = new
   GameDbContext(_dbContextOptions))
  {
```


WOW! eBook
www.wowebook.org

```
            return await dbContext.TwoFactorCodeModels.AnyAsync(
             x => x.TokenProvider == tokenProvider &&
             x.TokenCode == token && x.UserId == user.Id);
          }
        }
```

6. Go to the `Areas/Account/Views/Home` folder, and update the index view:

```
@inject UserManager<TicTacToe.Models.UserModel> UserManager
@{ var isTwoFactor
=UserManager.GetTwoFactorEnabledAsync(Model).Result;  ...   }
<h3>Account Details</h3>
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-6 col-md-6">
      <div class="well well-sm">
        <div class="row">
          ...
           <i class="glyphicon glyphicon-check"></i><text>Two
             Factor Authentication </text>
              @if (Model.TwoFactorEnabled)<a asp-
                action="DisableTwoFactor">Disable</a>
              else  <a asp-action="EnableTwoFactor">Enable</a>
           </div>
         ...
```

7. Add a new file called `_ViewImports.cshtml` to the `Areas/Account/Views` folder:

```
@using TicTacToe
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
@addTagHelper *, TicTacToe
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

8. Update the `UserService` class and the user service interface, and then add a new method called `EnableTwoFactor`:

```
public async Task<IdentityResult> EnableTwoFactor(string name, bool
enabled)
{
  try
  {
    var user = await _userManager.FindByEmailAsync(name);
    user.TwoFactorEnabled = true;
    await _userManager.SetTwoFactorEnabledAsync(user, enabled);
    return IdentityResult.Success;
  }
```


WOW! eBook
www.wowebook.org

```
    catch (Exception ex)
    {
      throw;
    }
  }
```

Add another method, `GetTwoFactorCode`:

```
public async Task<string> GetTwoFactorCode(string userName, string
tokenProvider)
{
  var user = await GetUserByEmail(userName);
  return await
_userManager.GenerateTwoFactorTokenAsync(user,tokenProvider);
}
```

9. Update the `SignInUser` method in `UserService` for supporting two-factor authentication, if it is enabled:

```
public async Task<SignInResult> SignInUser(LoginModel
 loginModel, HttpContext httpContext)
{
  ...
    if (await _userManager.GetTwoFactorEnabledAsync(user))
      return SignInResult.TwoFactorRequired;
    ...
  }
  ...
}
```

10. Go to the `Areas/Account/Controllers` folder, and update `HomeController`. Update the `Index` method and then add two new methods called `EnableTwoFactor` and `DisableTwoFactor`:

```
[Authorize]
public async Task<IActionResult> Index()
{ var user = await _userService.GetUserByEmail(User.Identity.Name);
  return View(user); }

[Authorize]
public IActionResult EnableTwoFactor()
{
  _userService.EnableTwoFactor(User.Identity.Name, true);
  return RedirectToAction("Index");
}

[Authorize]
public IActionResult DisableTwoFactor()
```


WOW! eBook
www.wowebook.org

```
{
  _userService.EnableTwoFactor(User.Identity.Name, false);
  return RedirectToAction("Index");
}
```

Note that we will explain the `[Authorize]` decorator/attribute later in this chapter. It is used to add access restrictions to resources.

11. Add a new model called `ValidateTwoFactorModel` to the `Models` folder:

```
public class ValidateTwoFactorModel
{
  public string UserName { get; set; }
  public string Code { get; set; }
}
```

12. Update the `AccountController`, and add a new method called `SendEmailTwoFactor`:

```
private async Task SendEmailTwoFactor(string UserName)
{
  var user = await _userService.GetUserByEmail(UserName);
  var urlAction = new UrlActionContext
  { Action = "ValidateTwoFactor", Controller = "Account",Values =
    new { email = UserName,
    code = await _userService.GetTwoFactorCode(user.UserName,
    "Email") },
    Protocol = Request.Scheme, Host = Request.Host.ToString() };

  var TwoFactorEmailModel = new TwoFactorEmailModel
  {  DisplayName = $"{user.FirstName} {user.LastName}", Email =
    UserName, ActionUrl = Url.Action(urlAction) };
  var emailRenderService = HttpContext.RequestServices.
    GetService<IEmailTemplateRenderService>();
  var emailService = HttpContext.RequestServices.
    GetService<IEmailService>();
  var message = await emailRenderService.RenderTemplate(
     "EmailTemplates/TwoFactorEmail", TwoFactorEmailModel,
       Request.Host.ToString());
  try{ emailService.SendEmail(UserName, "Tic-Tac-Toe Two Factor
   Code", message).Wait();  }
  catch { }
}
```


WOW! eBook
www.wowebook.org

Note that in order to call `RequestServices.GetService<T>();`, you must also add `using Microsoft.Extensions.DependencyInjection;` as you have done previously in other examples.

13. Update the `Login` method in `AccountController`:

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel loginModel)
{
  if (ModelState.IsValid)
  {
    var result = await _userService.SignInUser(loginModel,
     HttpContext);
    if (result.Succeeded)
    {
      if (!string.IsNullOrEmpty(loginModel.ReturnUrl)) return
        Redirect(loginModel.ReturnUrl);
      else  return RedirectToAction("Index", "Home");
    }
    else if (result.RequiresTwoFactor) await SendEmailTwoFactor
     (loginModel.UserName);
       return RedirectToAction("ValidateTwoFactor");
    else
      ModelState.AddModelError("", result.IsLockedOut ? "User is
       locked" : "User is not allowed");
    }
  return View();
}
```

14. Add a new view called `ValidateTwoFactor` to the `Views/Account` folder:

```
@model TicTacToe.Models.ValidateTwoFactorModel
@{ ViewData["Title"] = "Validate Two Factor";Layout =
"~/Views/Shared/_Layout.cshtml"; }
<div class="container">
  <div id="loginbox" style="margin-top:50px;" class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Validate Two Factor Code</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div class="text-center">
          <form asp-controller="Account"asp-
            action="ValidateTwoFactor" method="post">
            <div asp-validation-summary="All"></div>
```


WOW! eBook
www.wowebook.org

```
                <div style="margin-bottom: 25px" class="input-group">
                  <span class="input-group-addon"><i class="glyphicon
                   glyphicon-envelope
                       color-blue"></i></span>
                  <input id="email" asp-for="UserName"
                    placeholder="email address"
                           class="form-control" type="email">
                </div>
                  <div style="margin-bottom: 25px" class="input-group">
                  <span class="input-group-addon"><i class="glyphicon
                    glyphicon-lock
                           color-blue"></i></span>
                  <input id="Code" asp-for="Code"
                      placeholder="Enter your code" class="form-
                       control">   </div>
                  <div style="margin-bottom: 25px" class="input-group">
                   <input name="submit"class="btn btn-lg btn-primary
                     btn-block"
                           value="Validate your code" type="submit">
                   </div>
               </form>
              </div>
             </div>
            </div>
          </div>
        </div>
```

15. Add a new view called `TwoFactorEmail` to the `Views/EmailTemplates` folder:

```
@model TicTacToe.Models.TwoFactorEmailModel
@{
  ViewData["Title"] = "View";
  Layout = "_LayoutEmail";
}
<h1>Welcome @Model.DisplayName</h1>
You have requested a two factor code, please click <a
 href="@Model.ActionUrl">here</a> to continue.
```

16. Update the `UserService` class and the user service interface, and then add a new method called `ValidateTwoFactor`:

```
public async Task<bool> ValidateTwoFactor(string userName,
         string tokenProvider, string token, HttpContext
           httpContext)
{
  var user = await GetUserByEmail(userName);
  if (await _userManager.VerifyTwoFactorTokenAsync
    (user,tokenProvider, token))
```


WOW! eBook
www.wowebook.org

```
    {
      ...
    }
    return false;
  }
```

17. In the `ValidateTwoFactor` method, add the following actual code that performs the validation through identity claims:

```
if (await _userManager.VerifyTwoFactorTokenAsync(user,
    tokenProvider, token))
            {
                var identity = new ClaimsIdentity
               (CookieAuthenticationDefaults.Authentication
                Scheme);
                identity.AddClaim(new Claim(ClaimTypes.Name,
                 user.UserName));
                identity.AddClaim(new Claim(ClaimTypes.GivenName,
                 user.FirstName));
                identity.AddClaim(new Claim(ClaimTypes.Surname,
                 user.LastName));
                identity.AddClaim(new Claim("displayName", $"
                 {user.FirstName} {user.LastName}"));

                if (!string.IsNullOrEmpty(user.PhoneNumber))
                    identity.AddClaim(new Claim
                     (ClaimTypes.HomePhone, user.PhoneNumber));

                identity.AddClaim(new Claim("Score",
                 user.Score.ToString()));
                await httpContext.SignInAsync
                (CookieAuthenticationDefaults.
                 AuthenticationScheme,
                  new ClaimsPrincipal(identity), new
                  AuthenticationProperties
                 { IsPersistent = false });
                return true;
            }
```

18. Update `AccountController`, and then add two new methods for two-factor authentication validation:

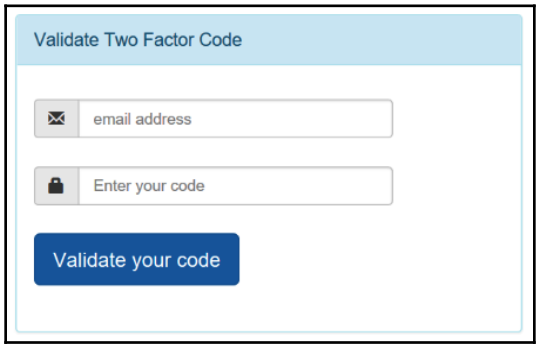
```
public async Task<IActionResult> ValidateTwoFactor(string email,
string code)
{
  return await Task.Run(() =>
  { return View(new ValidateTwoFactorModel { Code = code, UserName
=
```


WOW! eBook
www.wowebook.org

```
        email }); });
    }

    [HttpPost]
    public async Task<IActionResult>
    ValidateTwoFactor(ValidateTwoFactorModel validateTwoFactorModel)
    {
      if (ModelState.IsValid)
      {
        await _userService.ValidateTwoFactor(validateTwoFactorModel
         .UserName, "Email",
          validateTwoFactorModel.Code, HttpContext);
        return RedirectToAction("Index", "Home");
      }
      return View();
    }
```

19. Start the application, sign in as an existing user, and go to the **Account Details** page. Enable two-factor authentication (you might need to recreate the database and register a new user before this step):

20. Sign out as the user, go to the login page, and then sign in again. This time, you will be asked to enter a two-factor authentication code:



WOW! eBook
www.wowebook.org

21. You will receive an email with the two-factor authentication code:

22. Click on the link in the email and everything should be filled in for you automatically. Sign in and verify that everything is working as expected:

You can see how easy it is to implement two-factor authentication, as we did in the last section. Having gone through different forms of authentication, there will always be times when you may forget your password, and therefore we need to be able to reset our passwords securely so that we can be allowed back into our application after we re-authenticate our credentials. We will cover this in the next section.

Adding forgotten password and password reset mechanisms

Now that you have seen how to add authentication to your applications, you have to think about how you want to help users to reset their forgotten passwords. Users will always forget their passwords, so you need to have some mechanisms in place.

WOW! eBook
www.wowebook.org

The standard way of handling this type of request is to send an email reset link to the user. The user can then update their password, without the risk of sending the password in clear text through email. Sending a user password directly to a user email is not secure and should be avoided at all costs.

You will now see how to add a reset password feature to the Tic-Tac-Toe application:

1. Update the login form, and add a new link called `Reset Password Here` directly after the **Sign Up Here** link:

```
<div class="col-md-12 control">
  <div style="border-top: 1px solid#888; padding-top:15px;
   font-size:85%">
   Don't have an account?
   <a asp-action="Index"
    asp-controller="UserRegistration">Sign Up Here</a>
</div>
<div style="font-size: 85%;">
  Forgot your password?
  <a asp-action="ForgotPassword">Reset Password Here</a>
</div>
</div>
```

2. Add a new model called `ResetPasswordEmailModel` to the `Models` folder:

```
public class ResetPasswordEmailModel
{
  public string DisplayName { get; set; }
  public string Email { get; set; }
  public string ActionUrl { get; set; }
}
```

3. Update `AccountController`, and then add a new method called `ForgotPassword`:

```
[HttpGet]
public async Task<IActionResult> ForgotPassword()
{
  return await Task.Run(() =>
  {
    return View();
  });
}
```


WOW! eBook
www.wowebook.org

4. Add a new model called `ResetPasswordModel` to the `Models` folder:

```
public class ResetPasswordModel
{
  public string Token { get; set; }
  public string UserName { get; set; }
  public string Password { get; set; }
  public string ConfirmPassword { get; set; }
}
```

5. Add a new view called `ForgotPassword` to the `Views/Account` folder:

```
@model TicTacToe.Models.ResetPasswordModel
@{
  ViewData["Title"] = "GameInvitationConfirmation";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<div class="form-gap"></div>
<div class="container">
  <div class="row">
    <div class="col-md-4 col-md-offset-4">
      <div class="panel panel-default">
        <div class="panel-body">
          <div class="text-center">
            <h3><i class="fa fa-lock fa-4x"></i></h3>
            <h2 class="text-center">Forgot Password?</h2>
            <p>You can reset your password here.</p>
            <div class="panel-body">
              <form id="register-form" role="form"
               autocomplete="off" class="form"
               method="post" asp-controller="Account"
               asp-action="SendResetPassword">
                <div class="form-group">
                  <div class="input-group">
                    <span class="input-group-addon"><i
                     class="glyphicon glyphicon-envelope
                     color-blue"></i></span>
                    <input id="email" name="UserName"
                     placeholder="email address"
                     class="form-control" type="email">
                  </div>
                </div>
                <div class="form-group">
                  <input name="recover-submit"
                   class="btn btn-lg btn-primary btn-block"
                   value="Reset Password" type="submit">
                </div>
                <input type="hidden" class="hide"
```


WOW! eBook
www.wowebook.org

```
                                name="token" id="token" value="">
                            </form>

                          </div>
                        </div>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
```

6. Update the `UserService` class and the user service interface, and then add a new method called `GetResetPasswordCode`:

```
public async Task<string> GetResetPasswordCode(UserModel
    user)
{
  return await _userManager.
    GeneratePasswordResetTokenAsync(user);
}
```

7. Add a new view to the `View/EmailTemplates` folder called `ResetPasswordEmail`:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "View";
  Layout = "_LayoutEmail";
}
<h1>Welcome @Model.DisplayName</h1>
You have requested a password reset, please click <a
 href="@Model.ActionUrl">here</a> to continue.
```

8. Update `AccountController`, and then add a new method called `SendResetPassword`:

```
[HttpPost]
public async Task<IActionResult> SendResetPassword(string UserName)
{
  var user = await _userService.GetUserByEmail(UserName);
  var urlAction = new UrlActionContext
  {
    Action = "ResetPassword", Controller = "Account",
    Values = new { email = UserName, code = await
     _userService.GetResetPasswordCode(user) },
    Protocol = Request.Scheme, Host = Request.Host.ToString()
  };
```


WOW! eBook
www.wowebook.org

```
    var resetPasswordEmailModel = new ResetPasswordEmailModel
    {
      DisplayName = $"{user.FirstName} {user.LastName}", Email =
       UserName, ActionUrl = Url.Action(urlAction)
    };

    var emailRenderService = HttpContext.RequestServices.
      GetService<IEmailTemplateRenderService>();
    var emailService = HttpContext.RequestServices.
      GetService<IEmailService>();
    var message = await emailRenderService.RenderTemplate(
       "EmailTemplates/ResetPasswordEmail",
        resetPasswordEmailModel,Request.Host.ToString());
    try
    { emailService.SendEmail(UserName,"Tic-Tac-Toe Reset Password",
      message).Wait(); }
    catch { }

    return View("ConfirmResetPasswordRequest",
      resetPasswordEmailModel);
  }
```

9. Add a new view called `ConfirmResetPasswordRequest` to the `Views/Account` folder:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "ConfirmResetPasswordRequest";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
@section Desktop{<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<h1>@Localizer["You have requested to reset your password,
 an email has been sent to {0}, please click on the
provided
 link to continue.", Model.Email]</h1>
```

10. Update `AccountController`, and then add a new method called `ResetPassword`:

```
public async Task<IActionResult> ResetPassword(string email, string
code)
{
  var user = await _userService.GetUserByEmail(email);
  ViewBag.Code = code;
  return View(new ResetPasswordModel { Token = code, UserName =
email });
}
```


WOW! eBook
www.wowebook.org

11. Add a new view to the `Views/Account` folder called `SendResetPassword`:

```
@model TicTacToe.Models.ResetPasswordEmailModel
@{
  ViewData["Title"] = "SendResetPassword";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
@section Desktop{<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<h1>@Localizer["You have requested a password reset, an
email has been sent to {0}, please click on the link to
continue.", Model.Email]</h1>
```

12. Add a new view called `ResetPassword` to the `Views/Account` folder:

```
@model TicTacToe.Models.ResetPasswordModel
@{
  ViewData["Title"] = "ResetPassword";
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<div class="container">
  <div id="loginbox" style="margin-top:50px;"
    class="mainbox
   col-md-6 col-md-offset-3 col-sm-8 col-sm-offset-2">
    <div class="panel panel-info">
      <div class="panel-heading">
        <div class="panel-title">Reset your Password</div>
      </div>
      <div style="padding-top:30px" class="panel-body">
        <div class="text-center">
          <form asp-controller="Account"
           asp-action="ResetPassword" method="post">
            <input type="hidden" asp-for="Token" />
            <div asp-validation-summary="All"></div>
            <div style="margin-bottom: 25px" class="input-
            group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-envelope
               color-blue"></i></span>
              <input id="email" asp-for="UserName"
               placeholder="email address"
               class="form-control" type="email">
            </div>
            <div style="margin-bottom: 25px" class="input-
            group">
              <span class="input-group-addon"><i
               class="glyphicon glyphicon-lock
               color-blue"></i></span>
```


WOW! eBook
www.wowebook.org

```
                        <input id="password" asp-for="Password"
                         placeholder="Password"
                         class="form-control" type="password">
                      </div>
                      <div style="margin-bottom: 25px" class="input-
                       group">
                        <span class="input-group-addon"><i
                         class="glyphicon glyphicon-lock
                         color-blue"></i></span>
                        <input id="confirmpassword"
                         asp-for="ConfirmPassword"
                         placeholder="Confirm your Password"
                         class="form-control" type="password">
                      </div>
                      <div style="margin-bottom: 25px" class="input-
                       group">
                        <input name="submit"
                         class="btn btn-lg btn-primary btn-block"
                         value="Reset Password" type="submit">
                      </div>
                    </form>
                  </div>
                </div>
              </div>
            </div>
          </div>
```

13. Update the `UserService` class and the user service interface, and then add a new method called `ResetPassword`:

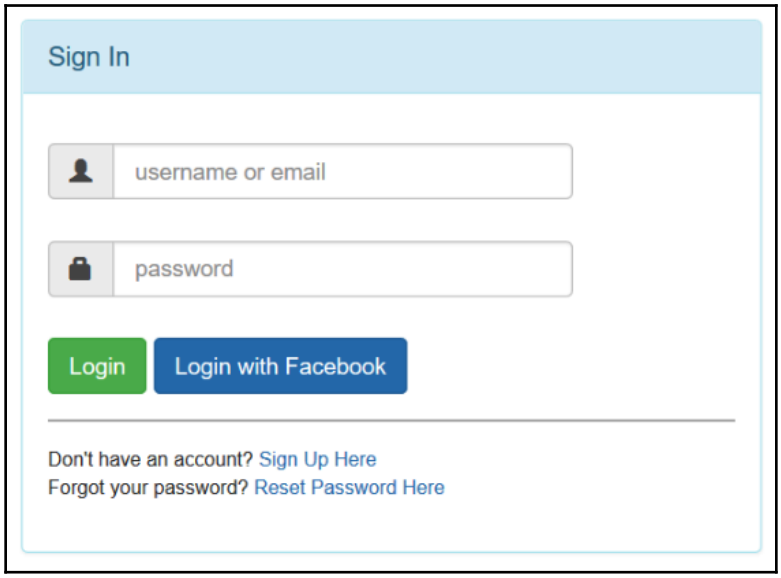
```
public async Task<IdentityResult> ResetPassword(string userName,
string password, string token)
{
  _logger.LogTrace($"Reset user password {userName}");
  try
  {
    var user = await _userManager.FindByNameAsync(userName);
    var result = await _userManager.ResetPasswordAsync(user, token,
     password);
    return result;
  }
  catch (Exception ex)
  {
    _logger.LogError($"Cannot reset user password {userName} -
     {ex}");
    throw ex;
  }
}
```


WOW! eBook
www.wowebook.org

14. Update `AccountController`, and then add a new method called `ResetPassword`:

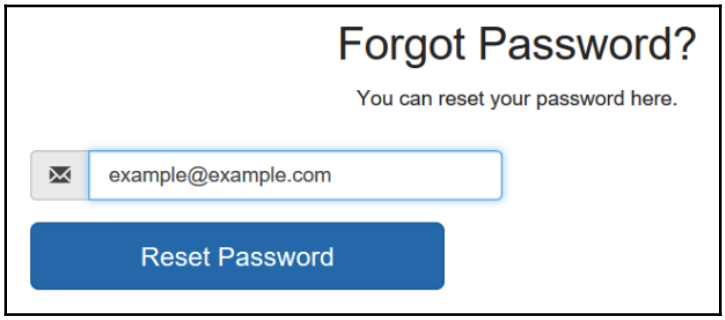
```
[HttpPost]
public async Task<IActionResult> ResetPassword(ResetPasswordModel
reset)
{
  if (ModelState.IsValid)
  {
    var result = await _userService.ResetPassword(reset.UserName,
                 reset.Password, reset.Token);
    if (result.Succeeded)
      return RedirectToAction("Login");
    else
      ModelState.AddModelError("", "Cannot reset your password");
  }
  return View();
}
```

15. Start the application and go to the login page. Once there, click on the **Reset Password Here** link:



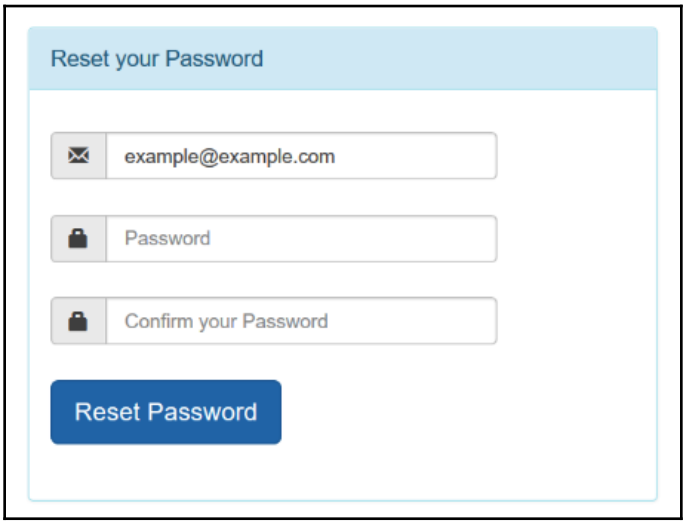
WOW! eBook
www.wowebook.org

16. Enter an existing user email on the **Forgot Password?** page; this will send an email to the user:



17. Open the **Password Reset** email and click on the link provided:

18. On the **Password Reset** page, enter a new password for the user and click on **Reset Password**. You should be automatically redirected to the **Login** page, so sign in with the new password:



WOW! eBook
www.wowebook.org

You will be excited to learn that we have now gone through all our authentication processes, and with the skills acquired, you are now able to provide reasonable authentication to any application that you may have your hands on. Now that a user is able to be authenticated, in other words, we know who our user is, we will not stop there.

Getting into the application does not necessarily mean that you are allowed to do anything that an application offers. We now need to know whether a user is authorized to do this or that action. And that's what we will look at in the next section.

Implementing authorization

In the first part of the chapter, you saw how to handle user authentication and how to work with user logins. In the next part, you will see how to manage user access, which will allow you to fine-tune who has access to what.

The simplest authorization method is to use the `[Authorize]` meta decorator, which disables anonymous access completely. Users need to be signed in to be able to access restricted resources in this case.

Now, let's go and see how to implement it within the Tic-Tac-Toe application:

1. Add a new method called `SecuredPage` to `HomeController`, and remove anonymous access to it by adding the `[Authorize]` decorator:

```
[Authorize]
public async Task<IActionResult> SecuredPage()
{
  return await Task.Run(() =>
  {
    ViewBag.SecureWord = "Secured Page";
    return View("SecuredPage");
  });
}
```

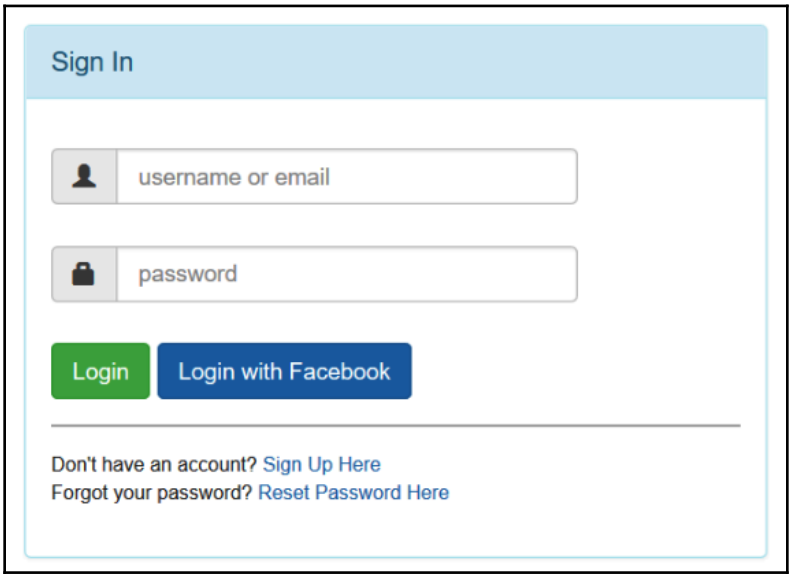
2. Add a new view called `SecuredPage` to the `Views/Home` folder:

```
@{
  ViewData["Title"] = "Secured Page";
}
@section Desktop {<h2>@Localizer["DesktopTitle"]</h2>}
@section Mobile {<h2>@Localizer["MobileTitle"]</h2>}
<div class="row">
  <div class="col-lg-12">
    <h2>Tic-Tac-Toe @ViewBag.SecureWord</h2>
```


WOW! eBook
www.wowebook.org

```
            </div>
        </div>
```

3. Try accessing the secured page by entering its URL, `http://<host>/Home/SecuredPage`, manually while not signed in. You will be redirected automatically to the **Login** page:



4. Enter valid user credentials and sign in. You should be automatically redirected to the secured page and now be able to view it:

Another relatively popular approach is to use role-based security, which provides some more advanced features. It is one of the recommended methods for securing your ASP.NET Core 3 web applications.

WOW! eBook
www.wowebook.org

The following example explains how to work with it:

1. Add a new class called `UserRoleModel` to the `Models` folder, and make it inherited from `IdentityUserRole<long>`. This will be used by the built-in ASP.NET Core 3 Identity authentication features:

```
public class UserRoleModel : IdentityUserRole<Guid>
{
  [Key]
  public long Id { get; set; }
}
```

2. Update the `OnModelCreating` method within `GameDbContext`:

```
protected override void OnModelCreating(ModelBuilder
 modelBuilder)
{
  ...
  modelBuilder.Entity<IdentityUserRole<Guid>>()
   .ToTable("UserRoleModel")
   .HasKey(x => new { x.UserId, x.RoleId });
}
```

3. Open the NuGet Package Manager Console and execute the `Add-Migration IdentityDb2` command. Then, execute the `Update-Database` command.
4. Update `UserService`, and modify the constructor to create two roles called `Player` and `Administrator`, if they do not yet exist:

```
public UserService(RoleManager<RoleModel> roleManager,
 ApplicationUserManager userManager, ILogger<UserService>
 logger, SignInManager<UserModel> signInManager)
{
  ...
  if (!roleManager.RoleExistsAsync("Player").Result)
    roleManager.CreateAsync(new RoleModel {
    Name = "Player" }).Wait();

  if (!roleManager.RoleExistsAsync("Administrator").Result)
    roleManager.CreateAsync(new RoleModel {
    Name = "Administrator" }).Wait();
}
```


WOW! eBook
www.wowebook.org

5. Update the `RegisterUser` method within `UserService`, and then add the user to the `Player` role or to the `Administrator` role during user registration:

```
...
try
{
  userModel.UserName = userModel.Email;
  var result = await _userManager.CreateAsync
    (userModel,userModel.Password);
  if (result == IdentityResult.Success)
  {
    if(userModel.FirstName == "Jason")
      await _userManager.AddToRoleAsync(userModel,"Administrator");
    else
      await _userManager.AddToRoleAsync(userModel, "Player");
  }
  return result == IdentityResult.Success;
}
...
```

Note that in the example, the code to identify whether a user has the administrator role is intentionally very basic. You should implement something more sophisticated in your applications.

6. Start the application and register a new user, and then open the `RoleModel` table within **SQL Server Object Explorer** and analyze its content:

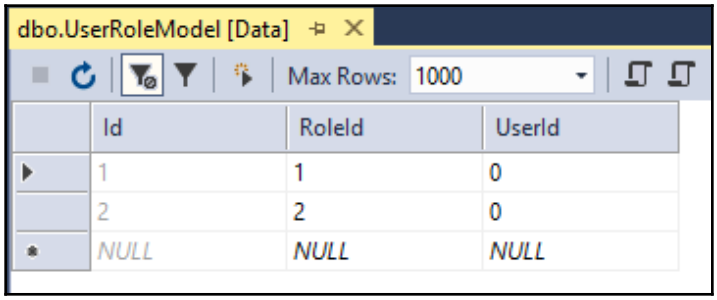
dbo.RoleModel [Data]

Max Rows: 1000

Id	ConcurrencyStamp	Name	NormalizedName
1	b3c69753-754e-405...	Player	PLAYER
2	bd1e92f8-9f4b-4e35...	Administrator	ADMINISTRATOR
NULL	NULL	NULL	NULL

WOW! eBook
www.wowebook.org

7. Open the `UserRoleModel` table within **SQL Server Object Explorer** and analyze its content:



8. Update the `SignInUser` method within `UserService` to map roles with claims:

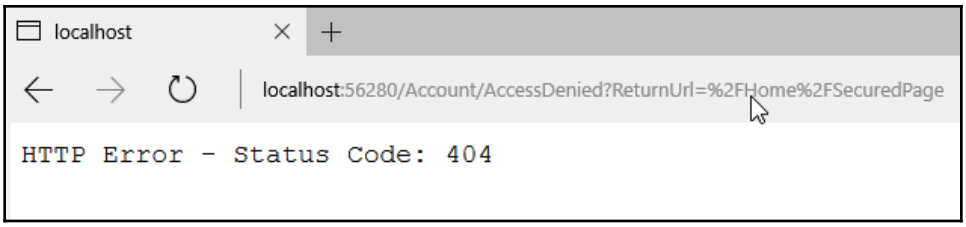
```
...
identity.AddClaim(new Claim("Score",
 user.Score.ToString()));
var roles = await _userManager.GetRolesAsync(user);
identity.AddClaims(roles?.Select(r => new
 Claim(ClaimTypes.Role, r)));

await httpContext.SignInAsync(
 CookieAuthenticationDefaults.AuthenticationScheme,
  new ClaimsPrincipal(identity),
  new AuthenticationProperties { IsPersistent = false });
...
```

9. Update the `SecuredPage` method within `HomeController`, use the administrator role to secure access, and then replace the `Authorize` decorator that was there initially with the following:

```
[Authorize(Roles = "Administrator")]
```

10. Start the application. If you try to access `http://<host>/Home/SecuredPage` without being logged in, you will be redirected to the **Login** page. Sign in as a user who has the player role, and you will be redirected to an **Access Denied** page (which does not exist, hence the **404** error) since the user does not have the administrator role:

WOW! eBook
www.wowebook.org



11. Log out and then sign in as a user who has the administrator role. You should now see the secured page, since the user has the necessary role:

In the following example, you will see how to sign in automatically as a registered user and how to activate claims-based and policy-based authentication:

1. Update the `SignInUser` method, and then add a new method called `SignIn` to `UserService`:

```
public async Task<SignInResult> SignInUser(LoginModel loginModel,
HttpContext httpContext)
{
  ...

    await SignIn(httpContext, user);

    return isValid;
  }
  catch (Exception ex)
  {
    ...
  }
  finally
  {
    ...
  }
}
```

Implement the `SignIn` method as follows:

```
private async Task SignIn(HttpContext httpContext, UserModel user)
{
  var identity = new ClaimsIdentity(CookieAuthenticationDefaults.
    AuthenticationScheme);
```


WOW! eBook
www.wowebook.org

```
    identity.AddClaim(new Claim(ClaimTypes.Name, user.UserName));
    identity.AddClaim(new Claim(ClaimTypes.GivenName,
      user.FirstName));
    identity.AddClaim(new Claim(ClaimTypes.Surname, user.LastName));
    identity.AddClaim(new Claim("displayName", $"{user.FirstName}
      {user.LastName}"));

    if (!string.IsNullOrEmpty(user.PhoneNumber))
      identity.AddClaim(new Claim(ClaimTypes.HomePhone,
        user.PhoneNumber));
    identity.AddClaim(new Claim("Score", user.Score.ToString()));

    var roles = await _userManager.GetRolesAsync(user);
    identity.AddClaims(roles?.Select(r => new Claim(ClaimTypes.
      Role, r)));

    if (user.FirstName == "Jason")
      identity.AddClaim(new Claim("AccessLevel", "Administrator"));

    await httpContext.SignInAsync(CookieAuthenticationDefaults.
      AuthenticationScheme,
       new ClaimsPrincipal(identity),new AuthenticationProperties {
          IsPersistent = false });
  }
```

Note that, in the example, the code to identify whether a user has administrator privileges is intentionally very basic. You should implement something more sophisticated in your applications.

2. Update the `RegisterUser` method in `UserService`, add a new parameter to automatically sign in a user after registration, and then re-extract the user service interface:

```
public async Task<bool> RegisterUser(UserModel userModel,
 bool isOnline = false)
{
  ...
  if (result == IdentityResult.Success)
  {
    ...
    if (isOnline)
    {
      HttpContext httpContext =
        new HttpContextAccessor().HttpContext;
      await Signin(httpContext, userModel);
    }
```


WOW! eBook
www.wowebook.org

```
  }
  ...
}
```

3. Update the `Index` method in `UserRegistrationController` to automatically sign in a newly registered user:

```
...
await _userService.RegisterUser(userModel, true);
...
```

4. Update the `ConfirmGameInvitation` method in `GameInvitationController` to sign an invited user in automatically:

```
...
await _userService.RegisterUser(new UserModel
{
  Email = gameInvitation.EmailTo,
  EmailConfirmationDate = DateTime.Now,
  EmailConfirmed = true,
  FirstName = "",
  LastName = "",
  Password = "Azerty123!",
  UserName = gameInvitation.EmailTo
}, true);
...
```

5. Add a new policy called `AdministratorAccessLevelPolicy` to the `Startup` class, just after the MVC Middleware configuration:

```
services.AddAuthorization(options =>
{
  options.AddPolicy("AdministratorAccessLevelPolicy",
   policy => policy.RequireClaim("AccessLevel",
   "Administrator"));
});
```

6. Update the `SecuredPage` method within `HomeController`, using `Policy` instead of `Role` to secure access, and then replace the `Authorize` decorator:

```
[Authorize(Policy = "AdministratorAccessLevelPolicy")]
```


WOW! eBook
www.wowebook.org

Note that it can be required to limit access to only one specific middleware since several kinds of authentication middleware can be used with ASP.NET Core 3 (cookie, bearer, and more) at the same time.
For this case, the `Authorize` decorator you have seen before allows you to define which middleware can authenticate a user.

Here is an example allowing cookies and a bearer token:

```
[Authorize(AuthenticationSchemes = "Cookie,Bearer",
 Policy = "AdministratorAccessLevelPolicy")]
```

7. Start the application, register a new user with an `Administrator` access level, sign in, and then access `http://<host>/Home/SecuredPage`. Everything should be working as before.

Note that you might need to clear your cookies and log in again to create a new authentication token with the required claims.

8. Try accessing the secured page as a user who does not have the required access level; as before, you should be redirected to `http://<host>/Account/AccessDenied?ReturnUrl=%2FHome%2FSecuredPage`:

9. Log out and then sign in as a user who has the `Administrator` role. You should now see the secured page since the user has the necessary role.

WOW! eBook
www.wowebook.org

Summary

In this chapter, you have learned how to secure ASP.NET Core 3 applications, including managing authentication and authorization for your application users.

You have added basic forms of authentication, and more advanced external provider authentication via Facebook, to the example application. This should give you some good ideas on how to approach these important topics in your own applications.

Furthermore, you have learned how to add standard reset password mechanisms, since users forget their passwords all the time and you need to respond to this type of request as securely as possible.

We have even talked about two-factor authentication, which can provide an even higher security level for critical applications.

At the end of the chapter, you also saw how to handle authorizations in multiple ways (basic, roles, policies), so that you can decide which approach is best suited to your specific use case.

In general, you have acquired the vital skills of being able to authenticate users for your application, and being able to authorize them to carry out assigned functions within the application.

In the next chapter, we will talk about the other different vulnerabilities you may have in developing ASP.NET Core 3 web applications.

WOW! eBook
www.wowebook.org

11
Securing ASP.NET Applications - Vulnerabilities

In the last chapter, we dealt with security mainly from the authentication and authorization point of view. We saw how to make sure that we know who is accessing our application and exactly what they are allowed to do within the application.

Unfortunately, unauthenticated logins and unauthorized access are not the only aspects that we need to guard against. In your quest as an application developer, you will be tasked with working on different applications of varying security importance. For apps that could motivate someone to actively seek ways in which they could exploit the application, then you, as a developer, need to make sure you can fend off potential hackers.

This chapter prepares you to be aware of the most common ways in which your web applications built with ASP.NET Core 3 could potentially be attacked.

For every application that you build, it is recommended that you look at its security right from the beginning and not only think about it at deployment time.

For some serious applications, as in the case of enterprise applications, it is not uncommon to even have a threat modeling session in which you try and analyze whatever possibilities there are in terms of threats to the application you are about to build, and then take those threats into consideration throughout the development phase.

In this chapter, you will learn different methods that malicious users usually use to exploit web applications and, apart from having an awareness, you will learn basic ways in which you can make sure that your application is safe from any would-be hackers.

WOW! eBook
www.wowebook.org

The following topics will be covered in this chapter:

- **Cross-Site Scripting** (**XSS**)
- Cookie stealing
- Eavesdropping, message tampering, and message replay
- Open redirects/XSR
- SQL injection
- **Cross-Site Request Forgery** (**XSRF/CSRF**)
- JS/JSON hijacking
- Over-posting
- Clickjacking
- Proper error reporting and stack trace

Cross-Site Scripting (XSS)

You will often find that **Cross-Site Scripting** is referred to, in its simplest form, as **XSS**, and it can be described as a form of HTML injection attack.

A website will be prone to an XSS attack if there are no measures in place to allow users' browsers to have scripts that could be executed. In this scenario, most of the time, the attacker assumes the identity of the user on the website and uses such a script to hijack an authentic user's session.

Once the session is in the hands of the attacker, then your application is at their mercy for the duration of the session. They can do just about anything, including making your web pages look any way they want and they can even launch attacks on other websites through your web pages. This can happen while an authentic user is still able to do other things, but an XSS attack can allow a hacker to assume full control of the browser.

If a website allows a user to upload links, then it is also susceptible to an XSS attack in which they would be able to harvest data uploaded through a form, and also be able to extract the website's security information.

XSS attacks can also come in the form of a hacker attempting to hijack cookies. These cookies can have identities for login and/or session identities. Once the cookies are hijacked, most information about the user is potentially available to the hacker. Through the same cookie hijacking, a hacker may ride on the user while performing normal functions to submit malicious content, such as scripts, without them being aware of such activity.

WOW! eBook
www.wowebook.org

Preventing XSS

Consider a common, real-life scenario where a user is redirected to a page that has been carefully designed by hackers to look exactly the same as an authentic application. This can happen in several ways, including through email content that has links that redirect you to a compromised URL. A key component that allows XSS attacks to thrive is when input data is insufficiently validated.

As a way of making sure that your input is thoroughly validated, you must make sure that you have a maximum length for any kind of input. You must make sure that the type of input into an input field is limited.

Always make sure that you filter out any possible Unicode characters for tags, for example, the less than and the greater than symbols: < and >.

HTML encoding that comes automatically with the `@` attribute in Razor is an effective way to prevent XSS attacks, but you still need to take extra steps to secure your site.

There is also a JavaScript encoder that you can inject into your views, as follows: `@inject JavaScriptEncoder jEncoder;` and then invoke the encoder directly for use, as in `@jEncoder.encode(...);`.

Upon receipt, ASP.NET Core 3 as a framework is always on guard to assess a request. It checks for scripts and/or any markup in the request, and throws an exception if it encounters content that feels suspect according to its pre-defined parameters.

Microsoft SDL (short for **Security Development Lifecycle**) guides you through security issues through the whole application development life cycle. It is explained in more detail at this link: `https://www.microsoft.com/en-us/securityengineering/sdl` and has several recommendations that you can follow.

Here's an example tip from the SDL: make sure to avoid using the `eval()` function in JavaScript, or indeed any similar functions that are meant to evaluate and subsequently execute a string input as a script, for example, `eval('2019+ 5')`.

WOW! eBook
www.wowebook.org

Cookie stealing

User experience is quite an important aspect of any web application. **Cookies** can play a part in having a website that enables a great user experience. There are many websites that actually use cookies to identify their users after they have logged in. On a website such as this, if you took out the cookies, you would have to log in again and again when navigating to different pages.

If a hacker can steal your cookies, they can easily pretend to be you. In this regard, you could be tempted to just disable the usage of cookies from your browser but, at the same time, there are many applications that force you to have them enabled.

Cookies could be used to store browsing history or site preferences, which are not all sensitive, but they can also have data that a website may utilize to identify you in between requests.

If a cookie used for authentication can be stolen, the user's identity can be assumed as well, therefore access is granted for all capabilities of the hijacked user. For this to be possible, though, the website must also be vulnerable to XSS, which was described earlier. The hacker can only steal a cookie if they are able to inject a script into the target website.

Preventing cookie stealing

You can tag your cookies with an `HttpOnly` attribute. This will make sure that a cookie that has this tag is only capable of being accessed by the server. This means that the cookie is safe from being accessed by any sort of script coming from the client side.

`HttpOnly` tagged cookies make it harder for a bulk of XSS attacks to succeed.

The `HttpOnly` attribute can possibly be set in `web.config`, just like in the following snippet:

```
<httpCookies domain="String" httpOnlyCookies="true" requireSSL="true">
```

The attribute could also be set individually for each cookie, like this:

```
Response.Cookies["CookieExample"].Value= "Value to be remembered";
Response.Cookies["CookieExample].HttpOnly=true;
```


WOW! eBook
www.wowebook.org

The `"CookieExample"` string is meant to contain a name of your choice that you assign to your cookie as a developer. Both `Value` and `HttpOnly` are attributes or properties for your named cookie that you can assign values to, as seen in the preceding example.

Eavesdropping, message tampering, and message replay

As implied in the heading, the vulnerabilities of **eavesdropping**, **message tampering**, and **message replay** are often explained as a **group**. This is because they are quite similar in the way that they behave and therefore are identified in the same way. They can also be prevented in similar ways.

Hackers might utilize a network data capture tool to record requests and responses from a client to a website. This is an example of eavesdropping.

If you do not put in place counter-measures against eavesdropping, a hacker could capture an HTTP request, modify it, and then submit it again to the website. This is what is now called message replay. This is clever on the part of the hacker because a website will be able to process the request, just like in a normal request, without raising any suspicions. This is because, in the case of a website that requires authentication, it usually has a required security token.

When we talk of message tampering, we mean that HTTP requests could be modified for malicious purposes, including to perform transactions and modify or even delete data.

Preventing eavesdropping and message replay

A commonly accepted way to prevent message replay in web applications using HTTP is by requiring communication to be carried out via the **Secure Sockets Layer** (**SSL**).

By using SSL in non-anonymous mode, you can protect your application from being instructed to replay messages back to the application server. If you do so, you also prevent HTTP request and response contents from being exposed to anyone listening in as an eavesdropper. Your guess is as good as mine as to whether SSL will also prevent message tampering.

WOW! eBook
www.wowebook.org

It is recommended that web applications use SSL in non-anonymous mode.

When connecting to a server through SSL, this is essentially how SSL works. A client checks that the identity of the server it is connecting to is correct by verifying that the server URI is the same as the hostname that is found in the SSL certificate.

There are times when a client may not have a certificate that may be used to verify the server. There are also times when a server uses protocols for SSL that do not have to have server identification. In both these instances, the SSL is being used in an anonymous mode.

You can choose, as you see fit, to configure anonymous SSL on some, but not other, web servers.

It must be noted that when one party poses as another during a client-server connection, then anonymous SSL cannot protect your application from spoofing threats or message replay. However, anonymous SSL can protect you from eavesdropping and tampering.

Open redirects/XSR

Open redirects, just as the name suggests, essentially redirect the user to a random website. These are also often referred to as **Cross-Site Redirects** (**XSR**) and they happen via your web application's URL.

Once a hacker is successful with a redirect, they can use it for a host of attacks, including spam and phishing. A hacker could also ride on your web application to serve malware to others.

XSR threats have more affinity toward web apps that make use of URL redirects through query strings and/or data in the form of an HTTP request.

Open redirects example

Here's a simplistic, real-world example of an open redirect at play. You might log into a website as a truly authentic user, but if a hacker has compromised a return URL, changing parts of the string after you have logged in will result in you being taken outside the application. You may not notice this as a user because the site that you have been redirected to could intentionally be created to look exactly like the original site.

WOW! eBook
www.wowebook.org

A compromise in the URL is harder to detect with longer URLs, in which just changing a single letter does the job of tricking you into thinking you are on the same site. A hacker with intent will have almost an exact replica of the authentic site and when you are on their compromised site, they might ask you to log in again for a made-up reason, and there goes your username and password!

Preventing open redirects

It is advisable, just to be on the safe side, to avoid any redirection from within your application. When you just have to have a redirect, then there is a helper method, `UrlHelper.IsLocalUrl()`, that you can use to make sure that you are only redirected to within the site.

With the `UrlHelper.IsLocalUrl()` method, you can make certain that a redirection goes to the same web server as the originating call, and never taken outside of your web application.

SQL injection

With regard to **SQL injection**, it's almost a given fact that any application will make use of queries against the database storage. Obviously, for any hacker, that presents an opportunity to utilize the queries for what they were not intended to do. They can do this by modifying the query for their intended purpose.

If you concatenate strings to make SQL statements and/or otherwise use dynamic SQL, this presents a particularly risky environment that can be exploited with SQL injection.

Preventing SQL injection

It doesn't matter what technology you are using to develop your application: all of them are susceptible to SQL injection. Therefore, you need to take steps to make sure that your application is safe from this kind of attack. Here is a recommendation: always use type-safe parameter encoding when constructing dynamic SQL statements.

In almost all data APIs, you will be allowed to specify exactly what type of parameter you are passing. This even includes ADO.NET as a technology, which has been around for a while. These parameters could be integers, Booleans, or other primitive types. Most data APIs provide for encoding or escaping as a way to guard against hacking attacks.

WOW! eBook
www.wowebook.org

Before you deploy your application into production, do a security audit on both your code and the application in general. Make sure that your database is locked down, with only the minimally required permissions for your application.

Protecting SQL connection strings

It's always vital to protect your connection string. It is recommended that you only put it as plain text in config or app settings. Storing it anywhere else in your code as plain text is asking for trouble. Through the **Microsoft Intermediate Language** (**MSIL**) disassembler, it is actually quite easy for anyone to see your connection string if you place it in code. A hacker can use the `Ildasm.exe` command to view your code's respective MSIL, through which the string will be laid bare.

Another aspect to consider is the fact that the different forms of connection strings do play a part. Some forms of connection strings can have a username and password; others just use the trusted connection or integrated security. If it is possible to do so, it is recommended to use the options that do not explicitly specify the username and password.

Desist from using a username and password for Windows authentication; rather, go for `Trusted_Connection = true` or `Integrated Security = SSPI`.

Using the Persist Security Info default value in connection strings

The default value for Persist Security Info is `False`. Setting it to `True` allows security-sensitive information, including the user ID and password, to be obtained from a connection after it has been opened. When set to `False`, security information is disposed of (after it has been used to open the connection), ensuring that any untrusted source does not have access to it.

Using object-relational mappers (ORMs)

Users of **object-relational mappers** (**ORMs**) such as Entity Framework Core 3, introduced in `Chapter 9`, *Accessing Data Using Entity Framework Core 3*, usually work with objects, and most ORMs offer strong, object-oriented query capabilities and therefore SQL injection is not as common a threat.

WOW! eBook
www.wowebook.org

Note that you could also use stored procedures along with Entity Framework Core. Usage of stored procedures further reduces the risk of SQL injection either when used alongside Entity Framework Core or on their own, mainly because of their parameterized features.

Even when string queries are used, ORMs usually make working with parameters so much easier than working with ADO.NET parameters, in that there is no drive to use string concatenation with most ORMs.

However, if you happen to use NHibernate, **HQL** (short for **Hibernate Query Language**) is very similar to SQL and it behaves in a similar manner as executing raw SQL statements.

If you are an NHibernate ORM user, desist from using HQL in your **Data Access Layer** (**DAL**) as it makes your application susceptible to SQL injection.

Cross-Site Request Forgery (XSRF/CSRF)

There are several mentions of SQL injection and XSS in books and blogs alike, but not too much is seen about the lesser-known **Cross-Site Request Forgery** threats, which can be equally devastating. In short form, it is referred to as either **XSRF** or **CSRF**.

In a nutshell, when you authentically log in to an application as a legitimate user, your identity can be exploited to be used to send requests to a compromised web application, which will carry out the requests with your identity.

Hackers can easily take advantage of XSRF/CSRF because of the concept of how the web itself is supposed to work in a stateless manner.

XSRF/CSRF is carried out in the form of a **confused deputy** attack. This means that an action can be fooled, unsuspectingly, by some other entity, but with a devastating result by misusing its legitimate authority.

XSRF/CSRF example

Let's see an example of a simple controller that may be susceptible to an XSRF/CSRF attack.

WOW! eBook
www.wowebook.org

At first glance, everything feels safe and secure but, in the following, we'll see how a controller with code such as this can be mouthwatering to an XSRF/CSRF hacker:

```
public class ContactController : Controller
{
   public ViewResult ContactDetails()
   { return View(); }
   public ViewResult Update()
   {
      Contact contact = DbContext.GetContact();
      contact.ContactId = Request.Form["ContactId"];
      contact.Name = Request.Form["Name"];
      SaveContact(contact);
      return View();
   }
}
```

Consider a scenario where a hacker sets up a page that deliberately targets this kind of controller. The hacker can then persuade a user to visit their page, which will then try to post to this controller. This controller will not be able to pick up the intended XSRF/CSRF attack when the user has already been authenticated via forms-based authentication or Windows authentication. See the following code:

```
<body onload="document.getElementById('contactForm').submit()">
  <form id="contactForm" action="http://.../Contact/Update"
    method="post">
   <input name="ContactId" value="123456" />
   <input name="Name" value="My Hack Example" />
  </form>
 </body>
```

This kind of attack is mitigated in different ways, as elaborated on in the next section.

Preventing XSRF/CSRF

The following are the most common ways through which XSRF/CSRF attacks can be thwarted.

Domain referrers

It is recommended to check and see whether an incoming HTTP request header referrer domain is indeed yours. When you do so, you can guard against any requests that are coming from potentially compromised sources from outside of your domain.

WOW! eBook
www.wowebook.org

This prevention method is not foolproof, though. If a user has Adobe Flash installed, hackers could actually take advantage and spoof the header. Some users may also actually decide not to send referrer headers as a deliberate choice for their privacy.

User-generated tokens

It is recommended to use a hidden HTML field to persist a token for a specific user, which is usually generated from the origin server, and then verify that the submitted token is valid. You can use a user's session or an HTTP cookie to keep the token that was generated, for later retrieval.

In ASP.NET Core MVC, short for Model View Controller, you can verify requests by creating a token for a specific user that is passed between the view and the controller. If the token is not the same, this is potentially an XSRF/CSRF attack and you can make provisions not to allow the request to continue. All this, as described, can be achieved by using an HTML helper that is available from within ASP.NET Core MVC, which is `@Html.AntiForgeryToken()`, used in a form that needs to be submitted and this is in the views section. For every request, this helper will add a hidden field named `RequestVerificationToken` with a token from the view that needs to be verified by the controller.

You can use the `AntiForgeryToken` functionality that is available with ASP.NET Core MVC to prevent XSRF/CSRF attacks.

This approach requires that the corresponding controller works in sync to make sure that it understands that the form data it is receiving contains an anti-forgery token. This is done by decorating the specific action in the controller with the `ValidateAntiForgeryToken` attribute. This attribute confirms that the HTTP request contains both a cookie value and the hidden form field, as mentioned previously, and verifies that these values are the same.

It is advisable to decorate every form with an anti-forgery token. This includes login forms.

WOW! eBook
www.wowebook.org

Limitations

As you have seen, anti-XSRF helpers, as described previously, are quite useful, but they have several limitations. If a user does not accept cookies on their browser, their requests will be rejected by the controller action decorated with `ValidateAntiForgeryToken`. You also need to make sure that your application is safe from XSS threats; otherwise, the anti-forgery token can be read. You must be mindful as well that the anti-forgery token does not work with HTTP `GET` requests, but only works with HTTP `POST` requests.

JS/JSON hijacking

Most modern web applications make use of JSON to pass data around. **JSON hijacking**, as its name suggests, is when a hacker accesses a JSON response from another site.

The motivation for a hacker to try and read a JSON response not meant for them is the fact that websites, usually, will contain a user's information that can personally identify someone in a JSON response. That's a gold mine for a would-be malicious user.

Preventing JSON hijacking

It is quite simple to prevent anyone from hijacking your JSON, mainly by making sure that you never design your APIs to return JSON arrays as an HTTP response.

You can also make use of an `HttpPost` attribute to decorate a specific action in your respective controller so that it should only give responses to HTTP requests that use an HTTP `POST` action.

Make sure that JSON services always return responses as non-array JSON objects.

Over-posting

ASP.NET Core MVC has a feature called **model binding**, which makes your life easy as a developer to be able to map your user inputs to a specific model automatically.

You can see the motivation for a hacker here to be able to piggyback on this feature to insert content into your model that a user did not actually fill in on a respective form.

WOW! eBook
www.wowebook.org

Vulnerability example

Let's use a hypothetical blog post page where users will be able to post their comments:

```
public class BlogComment
{
   public int CommentID { get; set; } // Primary key
   public int BlogPostID { get; set; } // Foreign key
   public BlogPost BlogPost { get; set; } // Foreign entity
   public string UserName { get; set; }
   public string Comment { get; set; }
   public bool IsApproved { get; set; }
}
```

Then we can have a form that needs to be visible to the blog reader:

```
Name: @Html.TextBox("UserName")
Comment: @Html.TextBox("Comment")
```

Now, in this case, our expectation is that we will be getting a blog user to only provide input to the `UserName` and `Comment` fields. A hacker with intent can use advanced browser tools to find out about, for example, the `IsApproved` Boolean field, and add it as part of the data to be posted or in the query string as `IsApproved=true`.

The feature that we are exploring, called model binding, typically does not know what fields were legitimately filled in, and will duly oblige and set `IsApproved` equal to `true`.

This is quite trivial for this example, but can you imagine if the model was, for example, `'student'`, and the hacker was able to simply set `student.Grade = 99`? Yes, that gives a lot of hope to most of us, but the point you need to note is the fact that this can have serious consequences. Think of a bank account with an `AvailableBalance` field that could easily be manipulated in that way. The importance of having counter-checks is eminent and explored in the next section.

Preventing over-posting

We can use the `BindProperty` attribute to decorate either a model or a specific controller action.

You can either have a blacklist or a whitelist approach in using the `BindProperty` attribute. A whitelist approach proves to be safer and simpler because you simply target those properties you need to bind.

WOW! eBook
www.wowebook.org

As another form of mitigation, we can just create a view model with just the properties that are needed for a user to fill in, and in that way prevent any binding targeted directly at your full model.

The usage of `BlogCommentViewModel` will be able to prevent over-posting:

```
public class BlogCommentViewModel {
    public string UserName { get; set; }
   public string Comment { get; set; }
}
```

Clickjacking

There are several synonyms associated with **clickjacking**, including **UI redress** attack and **UI redressing**. UI in these instances means user interface.

This vulnerability happens when a hacker compromises an application's links and buttons in such a way that users think they are clicking on a link/button to carry out, for example, function `A`, while, in reality, they are carrying out function `B`. This compromised function is decided by the hacker.

This attack happens mainly in browsers when a hacker has managed to embed a script into vulnerable links and/or buttons. The user clicks on those links with innocent intent but, in doing so, they are prone to either revealing sensitive information or the hacker taking full control of their computer.

Clickjacking example

You can have a scenario in which there is a button aimed at performing function `A`, but a hacker will place a replica button with a lower z-index that gets triggered to perform function `B` when a user has clicked the normal button that they are able to see.

Preventing clickjacking

We can use an HTTP response header called `x-frame-options` to combat clickjacking on our web application. We could do this in any of the following two implementations.

1. When your application starts up, you can place the following code in a module that implements `IHttpModule`.

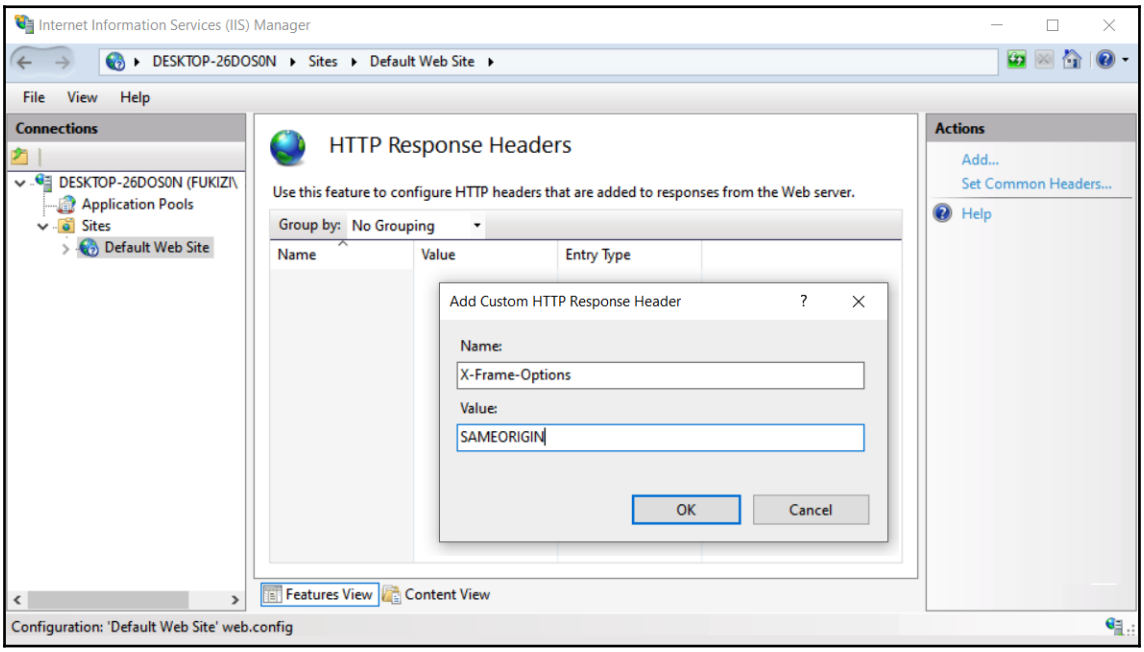
WOW! eBook
www.wowebook.org

Set the `x-frame-options` header to `"DENY"` when the web application starts up:

```
public void Init(HttpApplication application)
{
  application.BeginRequest += (new
   EventHandler(this.Application_BeginRequest));
}
private void Application_BeginRequest(object sender, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;
    context.Current.Response.AddHeader("x-frame-options", "DENY");
}
```

The main thing we are doing here is setting the `x-frame-options` HTTP header to `DENY`, but it is advisable to actually do this within your **Internet Information Services** (**IIS**) if this is your hosting option.

2. You could do so by going into the IIS Manager, and then with your website selected, do the following:
    1. Select or double-click to edit the HTTP response headers from the list of features.
    2. You will then click **Add...** from the **Actions** section on the right. See the following screenshot:



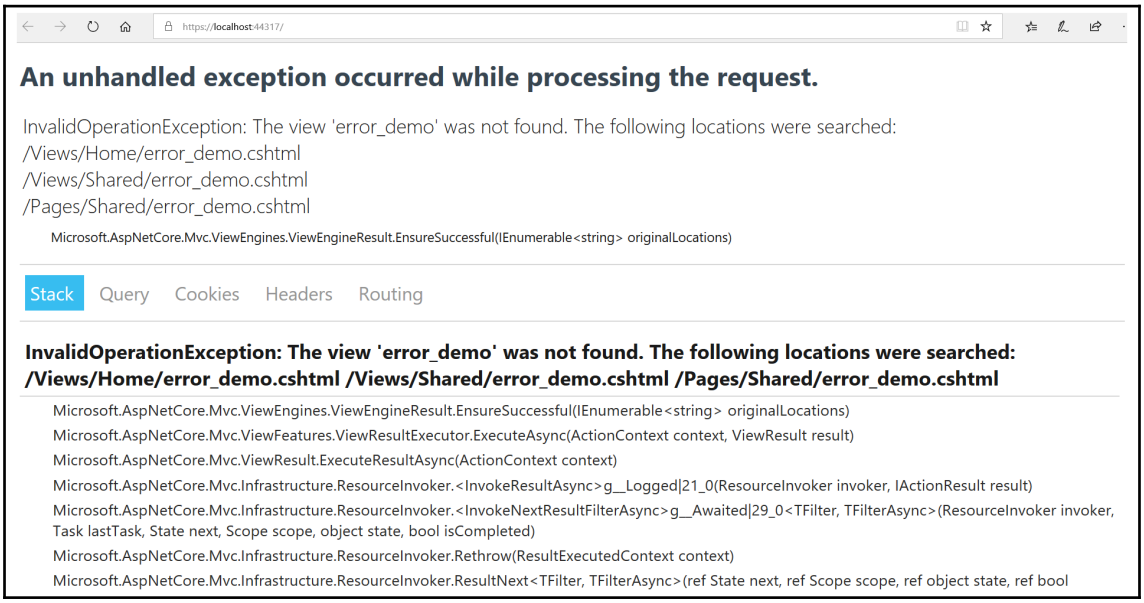
WOW! eBook
www.wowebook.org

3. A pop-up box will appear and then you can type `X-Frame-Options` in the **Name** input box, and `SAMEORIGIN` in the **Value** input box. Instead of `SAMEORIGIN`, you can also choose to type `DENY` and it will all serve the same purpose of protecting you from clickjacking.

Most of the previously mentioned potential attacks originate from an active hacking intent, devising ways and means to find loopholes to get into vulnerable web applications. However, as a web application developer with ASP.NET Core 3, the following vulnerability emanates from a bit of carelessness that you should always avoid.

Proper error reporting and stack trace

A **screen of death** sometimes referred to in its short form, **SOD**, could appear when an unhandled exception or error occurs in an ASP.NET Core 3 web application. An example is shown here:



This is sometimes referred to as a yellow screen of death because of the yellow color associated with these exception messages when you are using, say, either Google Chrome or Firefox as a browser.

WOW! eBook
www.wowebook.org

Error reporting vulnerability example

In its simplest terms, it is a bit careless for a web application developer to allow those kinds of errors to be seen by actual application users. The information that is contained on the screen of death only belongs to the application developer, not a visitor to your application. For a determined hacker, this information will give them somewhere to start as it gives out internal information on the application that may give a lead and an insight into how your application actually works.

Error handling needs to follow a proper thought process. This applies even in seemingly trivial situations where an application developer needs to use safe casting and proper type conversions such as `TryParse`, as this goes a long way in preventing ad hoc errors that can result in a screen of death.

Preventing a screen of death

There is a simple solution that will prevent your application from producing a screen of death, and that is to simply make sure that you configure and specify a custom error page. Luckily, ASP.NET Core 3 provides ready-made plumbing for you to achieve that through middleware in the `Startup` class.

It's not a nice experience on the part of the user to see a screen of death. It is better to think of your user and give them a friendly error page when something goes wrong.

You can specify the custom error page in the `Startup` class of the `Configure` method:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

This determines what kind of error page will be shown to the user according to the environment, whether it is a development or other kind of environment, including production.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we have looked at the common vulnerabilities that we need to be aware of when developing software applications with ASP.NET Core 3. If we are going to effectively build a secure solution, it is quite important to have an idea from what angle a malicious attack is going to come from.

We looked at XSS attacks, where a malicious user piggybacks on an authentic user's identity with the aim of injecting scripts into HTML. We saw that one of the ways a hacker can gain a user's identity is by cookie stealing, which we can prevent by tagging our cookies with an `HttpOnly` attribute.

We looked at eavesdropping, message tampering, and message replay using network gadgets, and we also had a look at open redirect/XSR attacks, which redirect a user to external malicious websites. We looked at SQL injection, XSRF/CSRF, JS/JSON hijacking, over-posting, and clickjacking. We also saw how important it is to do proper error reporting.

After learning about possible attacks and learning how to make sure that we are prepared for those kinds of attacks, we are now in a good position to put our application into production, where it will be exposed to public users. It's a good time to move on to the next chapter, in which we will be deploying and hosting our secured ASP.NET Core 3 application. Stay focused.

WOW! eBook
www.wowebook.org

12
Hosting ASP.NET Core 3 Applications

That's it: we are almost at the end of the book, which means that we have nearly finished the entire application development life cycle, and thus, customers will be able to use your applications soon! Be proud, because after reading and understanding this penultimate chapter of the book, you will have acquired strong skills to create and deploy your own mind-blowing applications with strong technical foundations!

Let's recap: from the beginning of the book until now, you have seen how to set up a development environment, how to use the various features of ASP.NET Core 3 to develop modern web applications, how to connect them to a database via Entity Framework Core, and, finally, in the previous two chapters, how to secure them against any malicious cybercriminals.

Now, we need to talk about the last step in the development cycle, which consists of hosting and deploying your applications once they are production-ready.

We will first look at the general hosting of our application, before we go into the specifics of deploying our application into the cloud, on two of the most common platforms: **Amazon Web Services** (**AWS**) and Microsoft Azure; and finally, we will look at using Docker containers to host our applications.

The goal of this chapter is to explain the different options you have in hosting your application, how to choose the right ones, and how to deploy your ASP.NET Core 3 web applications, using the most current technologies and cloud providers.

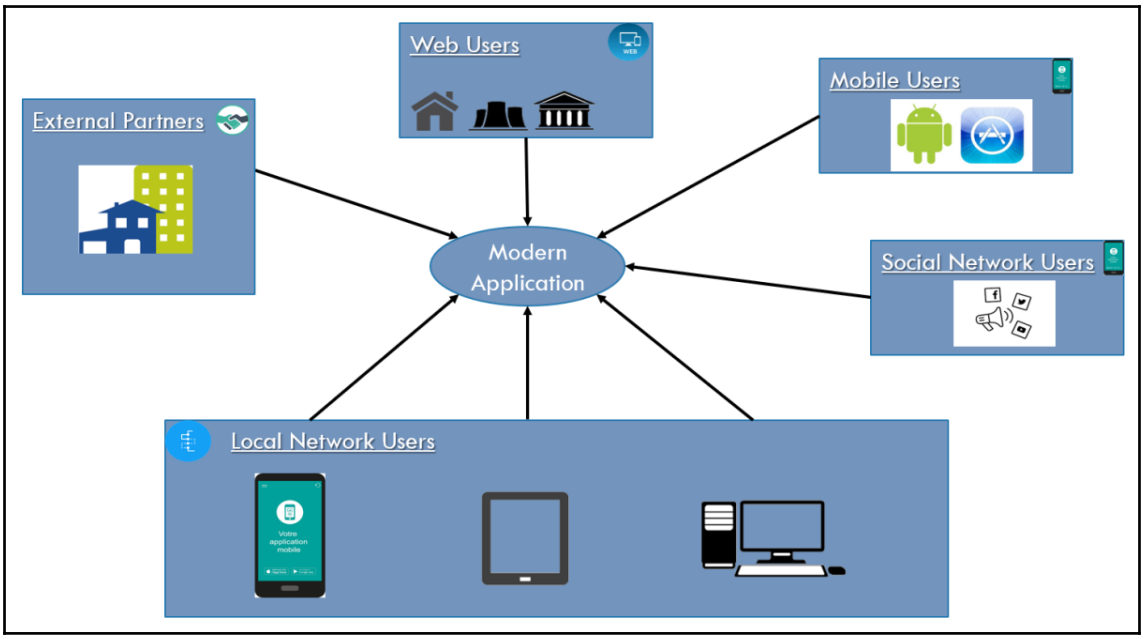
In this chapter, we will cover the following topics:

- Hosting applications
- Deploying applications in AWS
- Deploying applications in Microsoft Azure
- Deploying applications into Docker containers

WOW! eBook
www.wowebook.org

Hosting applications

You can build the best and most useful applications in the world, but if your customers cannot access them easily and from any device, you may not get the success you expect. As you can see in the following diagram, applications increasingly need to be omnichannel, which means customers need to be able to start on one device and then continue on another:



Your applications need to be deployable to multiple targets and, in some cases, multiple operating systems, to allow a high degree of flexibility and device availability. This is where hosting comes into play.

A host is responsible for application startup and lifetime management, which includes providing and configuring a server, and request processing. Depending on how you are hosting your ASP.NET Core 3 applications, you can support different devices for your applications. The selected technology has a significant impact on the possible choice of device and operating system.

ASP.NET Core 3 fully supports all current hosting mechanisms on multiple platforms and operating systems. It all depends on your specific application context.

WOW! eBook
www.wowebook.org

Some hosting examples for your ASP.NET Core applications are as follows:

- Host on Windows via **Internet Information Services** (**IIS**)
- Host in a Windows service
- Host on Linux, using NGINX
- Host on Linux, using Apache

During development time, or if you don't need to share your applications with others, it may be interesting to use self-hosting mechanisms or IIS Express, which provides a quick and easy solution for disconnected, **proof of concept** (**PoC**), or test projects.

However, if you start sharing your applications with others, you need more sophisticated hosting solutions and the corresponding server technologies.

For example, to expose your ASP.NET Core applications over the internet, you will need a web server that is accessible outside of your local network. There are several possible solutions to achieve this goal, and here are two of them:

- One is using an internet host provider to host your web server. However, you will need size and will need to manage the server by yourself, which may be expensive and time-consuming.
- Another option is to use public cloud providers, which offer much more flexibility and scalability while allowing cost reduction, as you only pay for what you need. The most famous ones are AWS and Microsoft Azure, which have data centers all around the world.

Furthermore, when using public cloud **Platform as a Service** (**PaaS**) offers, you don't even have to manage the operating system or the platform anymore. The cloud platform does everything for you. Instead, you can access cloud services, which provide web server or database server functionalities with high **service-level agreements** (**SLAs**). Two examples are AWS Elastic Beanstalk and Microsoft Azure App Service.

After having seen the various hosting options at your disposal, you will be able to decide on your deployment targets. For publicly available web applications, you will want to deploy to a public cloud provider. The next sections will show you how to deploy to the most common and famous public cloud providers, and how to use the most recent technologies to do so.

WOW! eBook
www.wowebook.org

Deploying applications in AWS

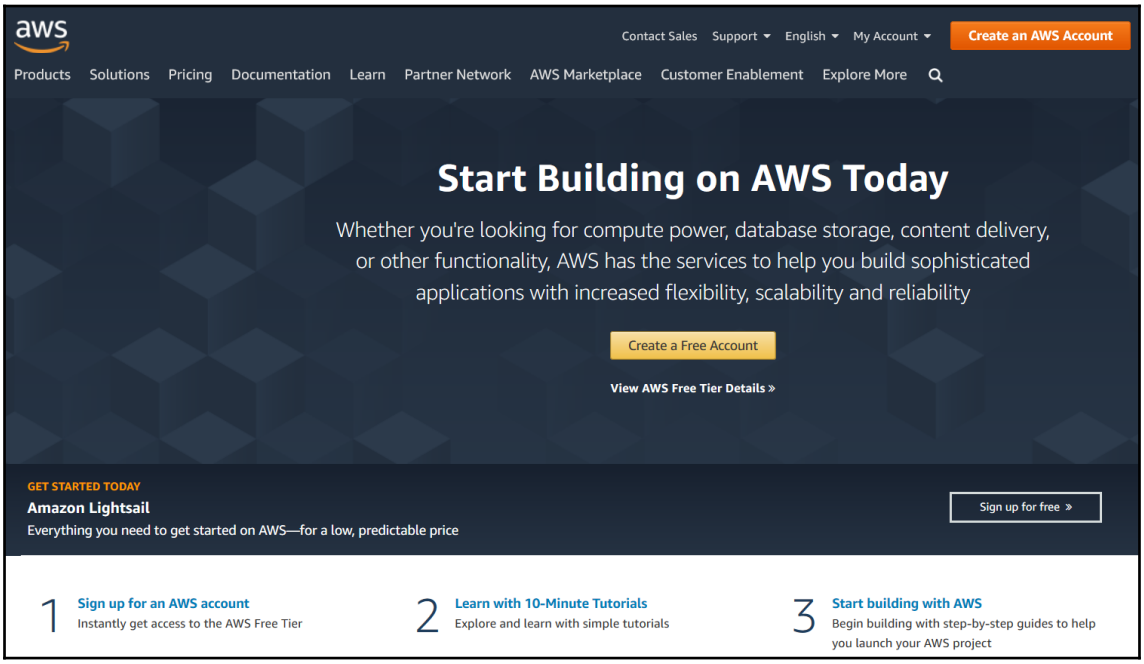
AWS, a subsidiary of Amazon.com, Inc., provides a public cloud computing platform for building, testing, deploying, and managing applications and services within globally available AWS data centers all around the world. It supports many different programming languages, tools, frameworks, and systems.

We will explore AWS in this section, and will see how to create an account and deploy your ASP.NET Core 3 applications to AWS Elastic Beanstalk.

First, you have to sign up for an account on AWS; it only takes five minutes, but you will need a credit card for this.

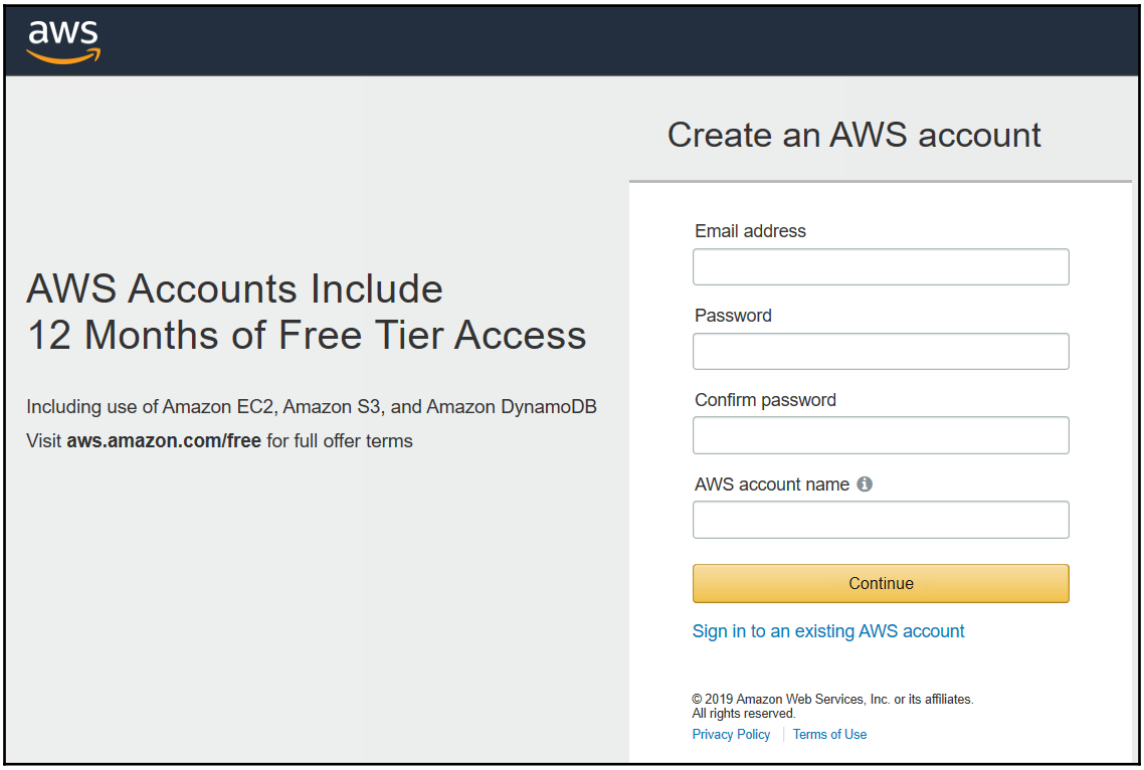
Let's go through the account registration steps, as follows:

1. Open a browser, go to `https://aws.amazon.com`, and click on the **Create a Free Account** button, as shown in the following screenshot:



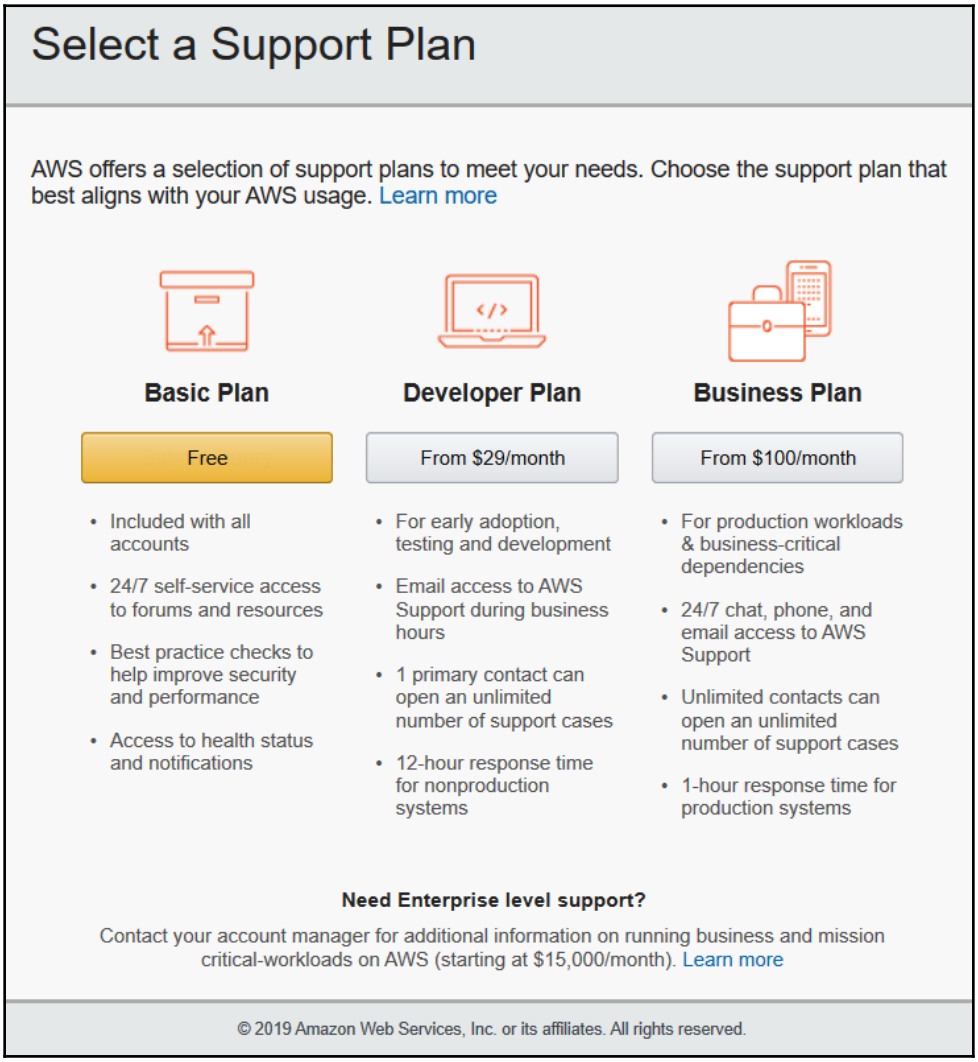
WOW! eBook
www.wowebook.org

2. Fill in the **Create a new AWS Account** form, continue with filling in contact information as well, and then click on **Continue**, as shown in the following screenshot:



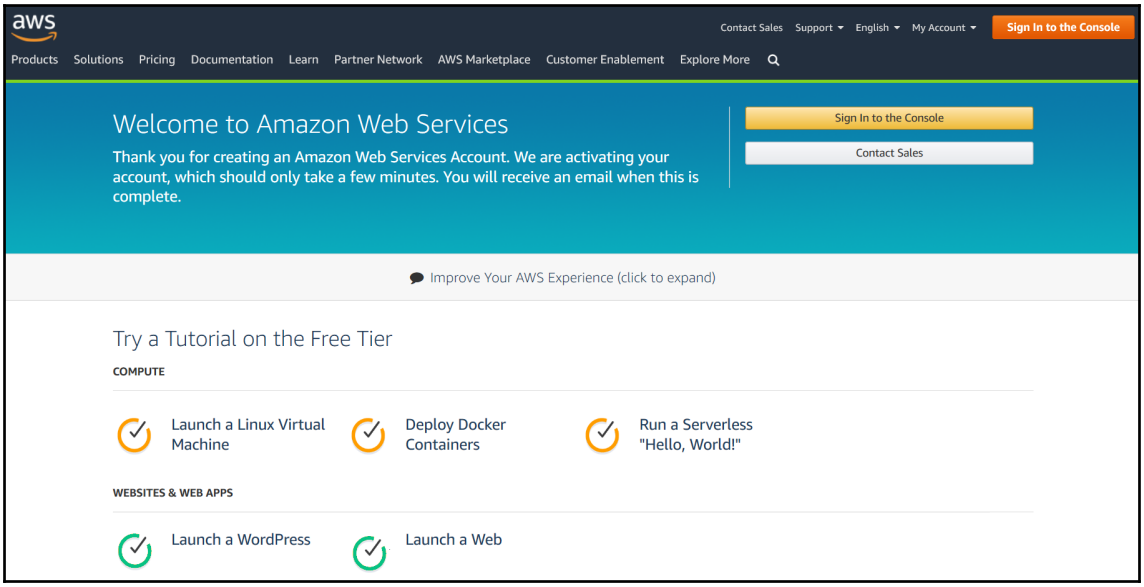
WOW! eBook
www.wowebook.org

3. Fill in the **Payment Information**, then click **Continue**. Fill in the **Identity Information Verification** form and click on **Continue**, then select a support plan and click on **Continue**, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

4. After the registration has been done, you are automatically redirected to the welcome page, where you should click on the **Sign In to the Console** button, as shown in the following screenshot:



After having created your new AWS user account, you are now ready to deploy your first ASP.NET Core application in AWS.

When working with AWS, you basically have the following two choices in terms of deploying your ASP.NET Core web applications:

- AWS Elastic Beanstalk
- AWS EC2 Container Service

The next section will shed some light on how to deploy your applications in AWS Elastic Beanstalk. So, stay tuned, engage your seat belt, and enjoy the ride!

Deploying applications in AWS Elastic Beanstalk

AWS Elastic Beanstalk is a PaaS offering for web-based applications in AWS that includes Auto Scaling. In this regard, it is comparable to Microsoft Azure App Service, which you will see in a later section of this chapter.

WOW! eBook
www.wowebook.org

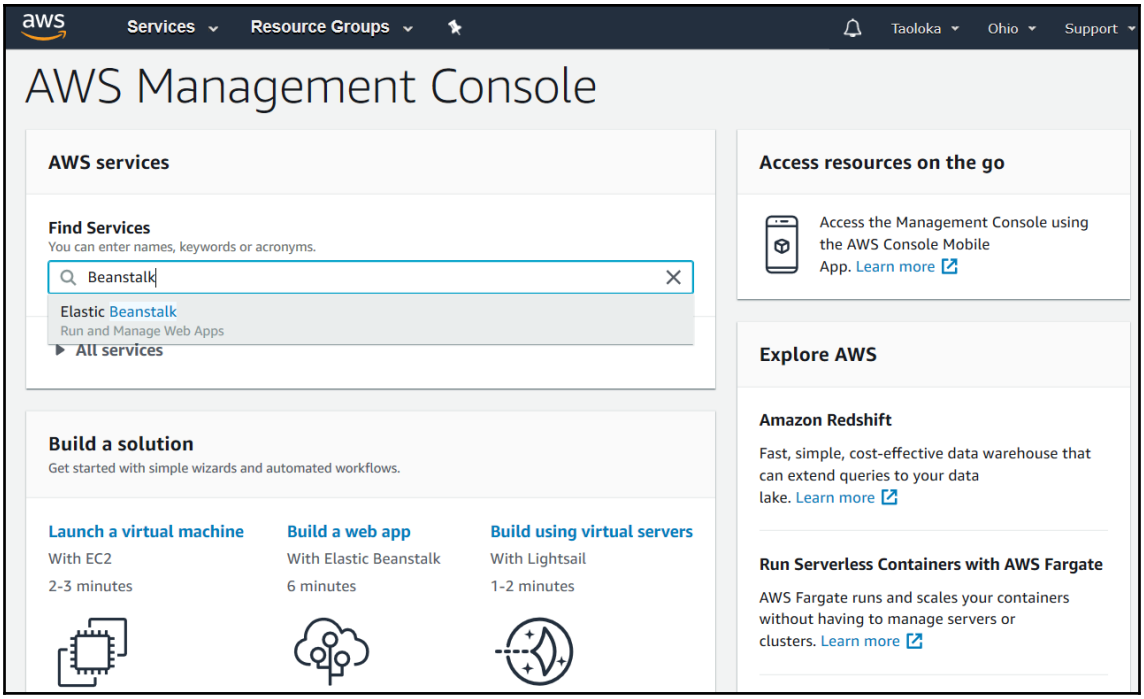
AWS Elastic Beanstalk removes the need to manage infrastructure and, instead, you only need to be concerned about building and hosting your applications. For a full DevOps approach, it is advised that you use this PaaS service if you want to work with AWS.

For more information on AWS Elastic Beanstalk, check out `https://aws.amazon.com/elasticbeanstalk/`.

The following instructions illustrate step by step how to deploy the Tic-Tac-Toe application in AWS Elastic Beanstalk.

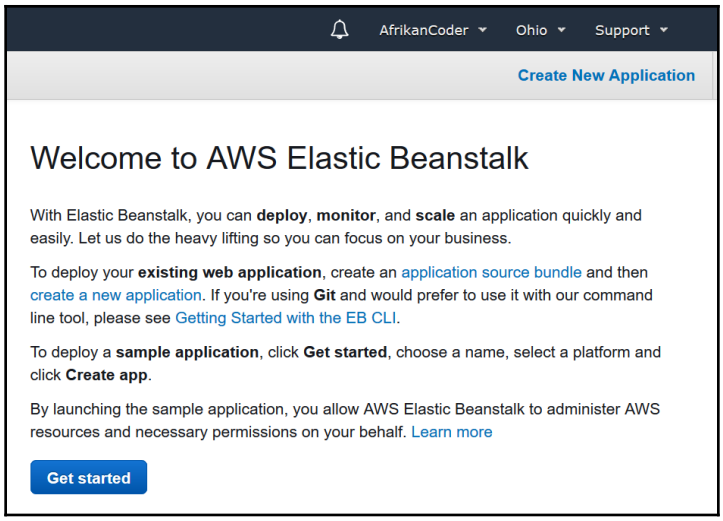
Let's start with the creation of the AWS Beanstalk application, as follows:

1. Sign in to AWS and go to the **AWS Management Console**, enter `Beanstalk` in the **AWS services** textbox, and click on the displayed link; you will be redirected to the Beanstalk welcome page, as follows:

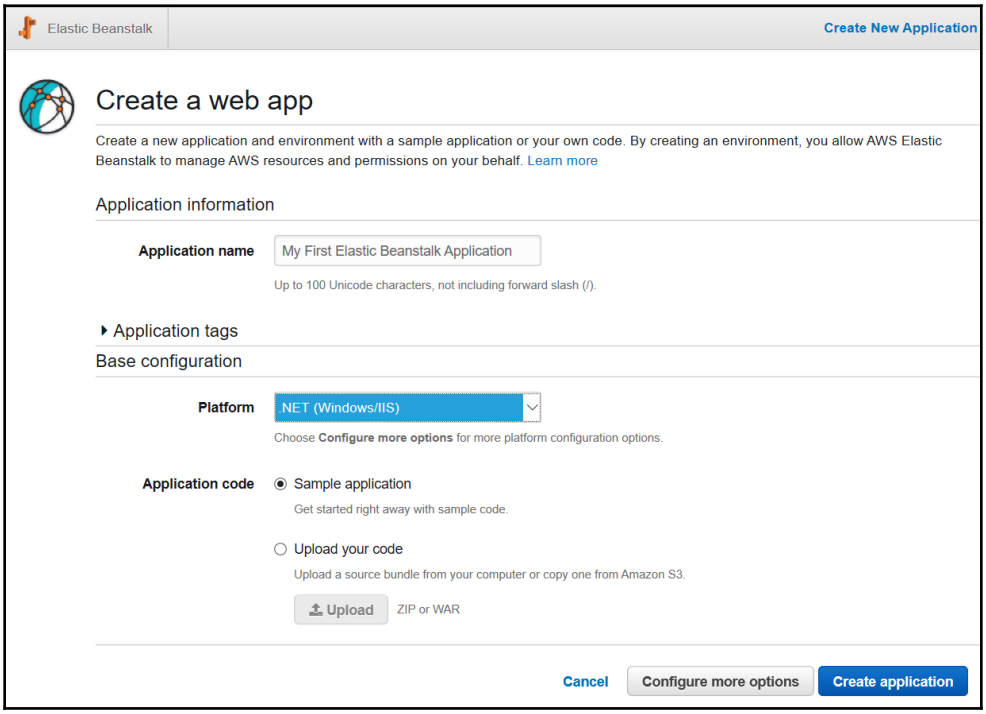


2. On the Beanstalk welcome page, click on the **Get started** button, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



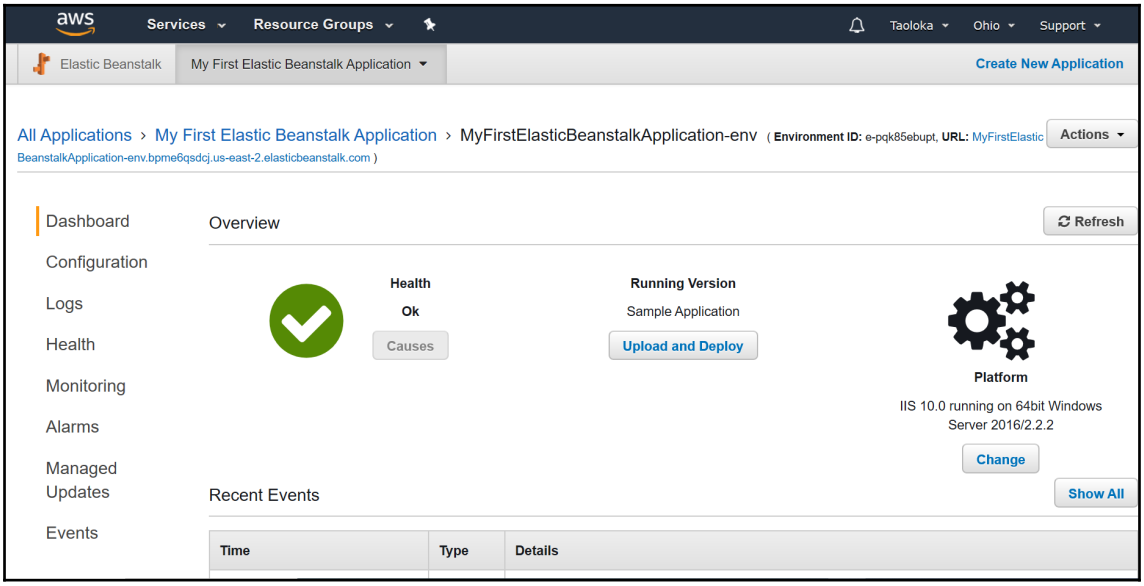
This will take you to the page displayed in the following screenshot. Select **.NET (Windows/IIS)** as a platform, then click on the **Create application** button, as follows:



WOW! eBook
www.wowebook.org

Note that you can change the IIS version and network settings (Network Load Balancer or single instance) by clicking on the **Configure more options** link, shown in the preceding screenshot.

3. Wait until the Beanstalk application has been created (depending on your internet connection and AWS, this may take a while), as follows:

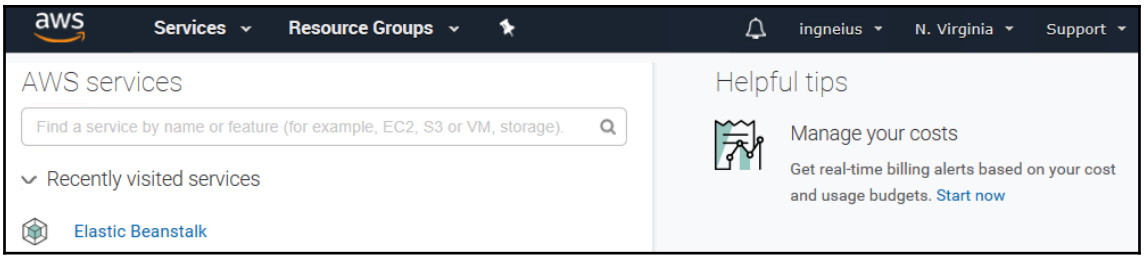


The technical environment needs to be prepared in the next steps, before being able to deploy the Tic-Tac-Toe application and then run it in the end.

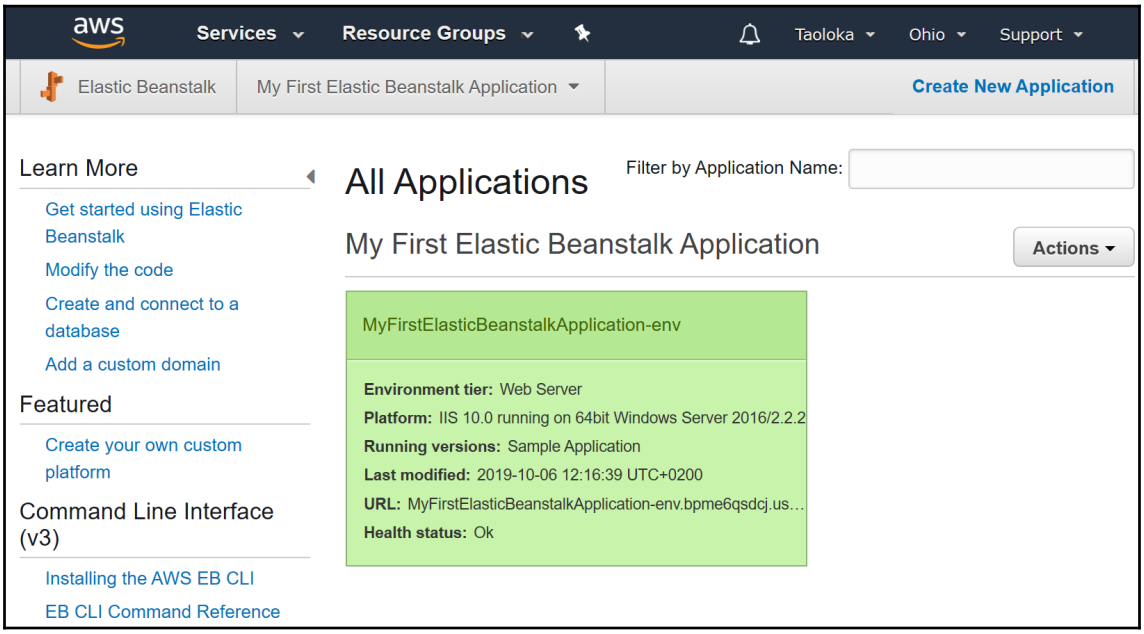
As you may have seen in the preceding chapters, the application requires a database to persist user and application data. For this purpose, we will make provision for a SQL Server PaaS service called the **Amazon Relational Database Service** (**Amazon RDS**) in AWS, as in the following example:

1. Return to the **AWS Management Console** and click on **Elastic Beanstalk** within the **Recently visited services** section, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

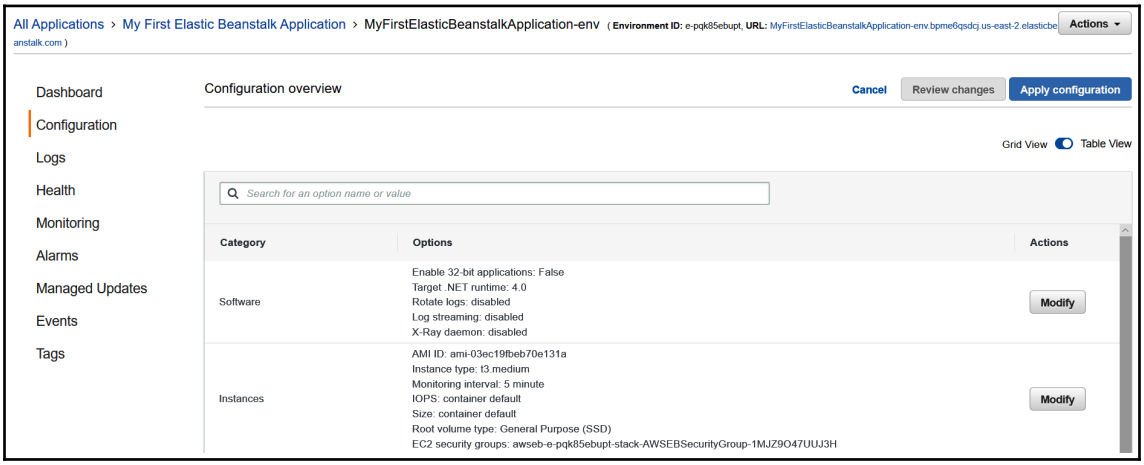


2. On the Beanstalk **All Applications** page, select the desired environment and then click on **Default-Environment**, and you will see the following resulting screen:



WOW! eBook
www.wowebook.org

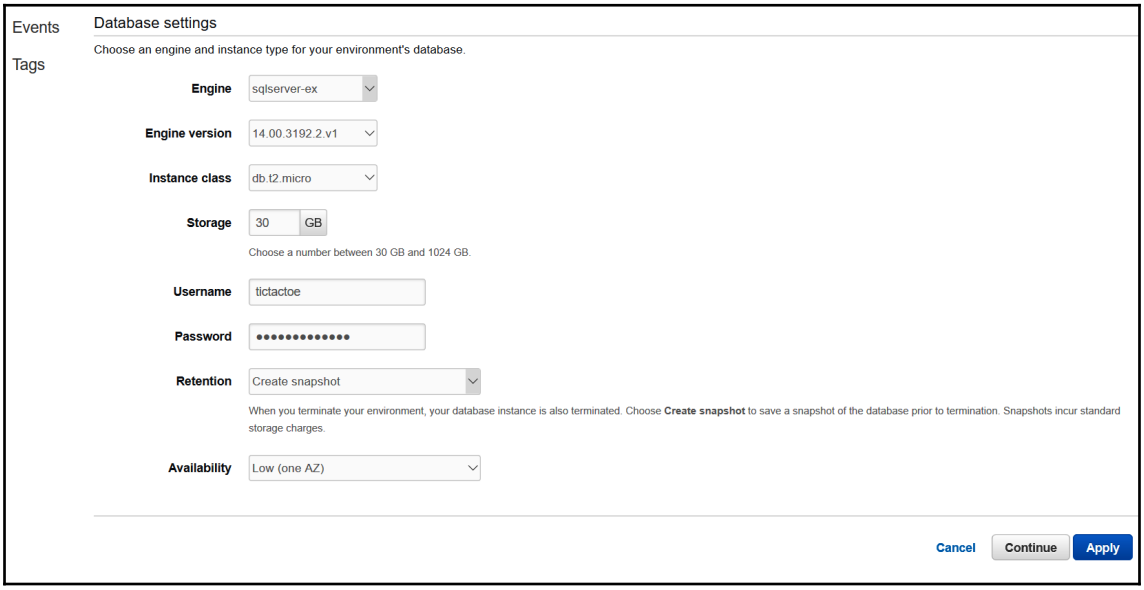
3. On the specific Beanstalk application page, click on **Configuration** in the left-hand menu, as shown in the following screenshot:



4. Scroll down, and click on the **Modify** database link, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

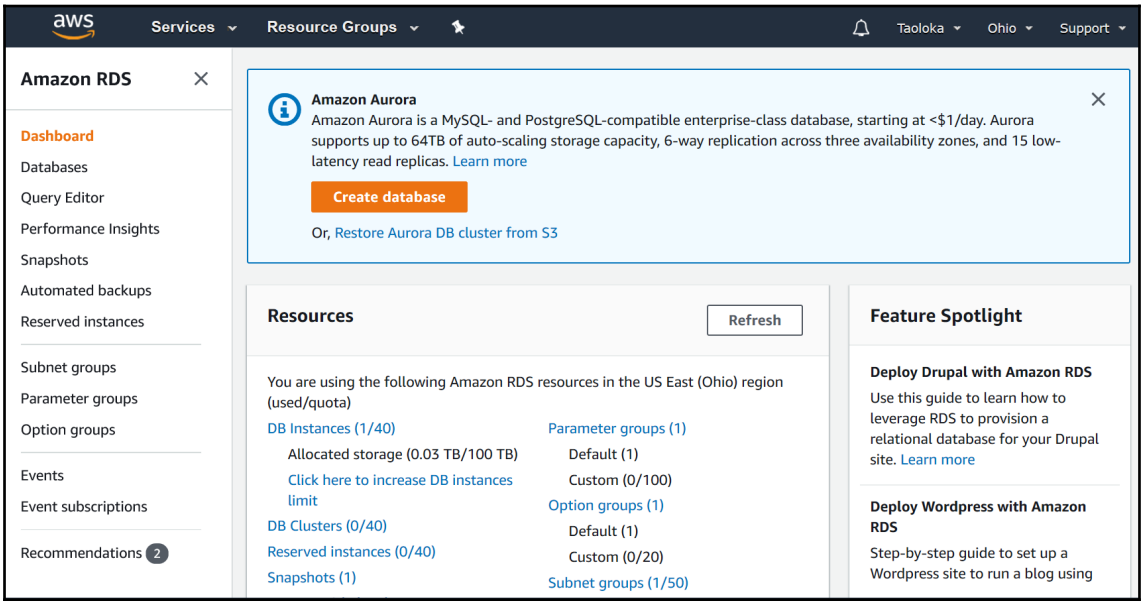
5. Select as **DB Engine** SQL Server Express (**sqlserver-ex**) and enter a master username and password; leave the rest of the fields at their default values, click on the **Apply** button at the bottom of the page, and wait for the database creation to be finished, as shown in the following screenshot:



Note that, depending on your application's needs, the SQL Server Express edition may not be enough since it is limited in size, meaning that the Enterprise or Web editions may be necessary, which will result in higher cloud-provider costs. For the Tic-Tac-Toe sample application, it is, however, largely sufficient.

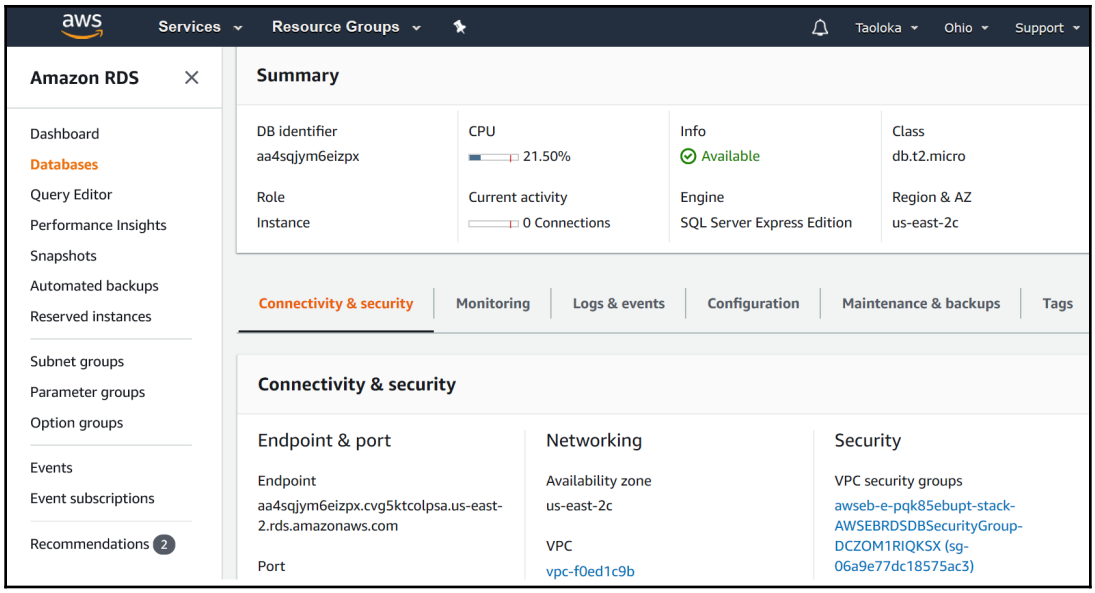
WOW! eBook
www.wowebook.org

6. Go to the **AWS Management Console**, enter `RDS` in the **AWS services** textbox, and click on the displayed link. You will be redirected to the **Amazon RDS** page; click on **DB Instances** in the **Resources** menu, as shown in the following screenshot:

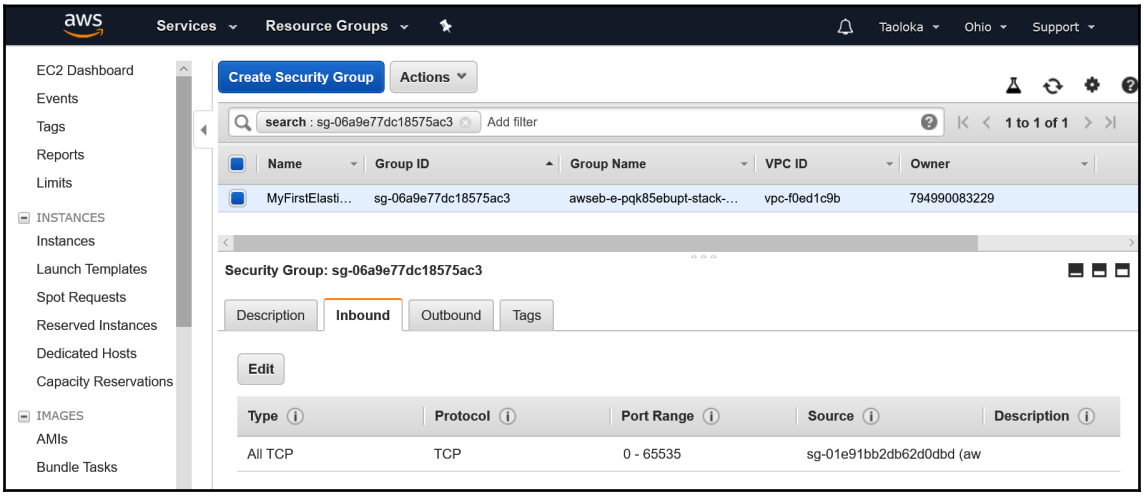


7. Click on your instance, and the instance dashboard will be displayed. Scroll down to retrieve the endpoint address, which will be used to update the application connection string before deployment, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



8. Further to the right, in the **Connectivity & security** tab on the **Amazon RDS Instance page**, click on the **VPC security groups** link, as shown in the preceding screenshot.
9. On the **Security Group** page, click on **Inbound** tab in the menu at the bottom of the page, then click on **Edit**, to be able to update the inbound rules for the security group of the database you have just created, as shown in the following screenshot:



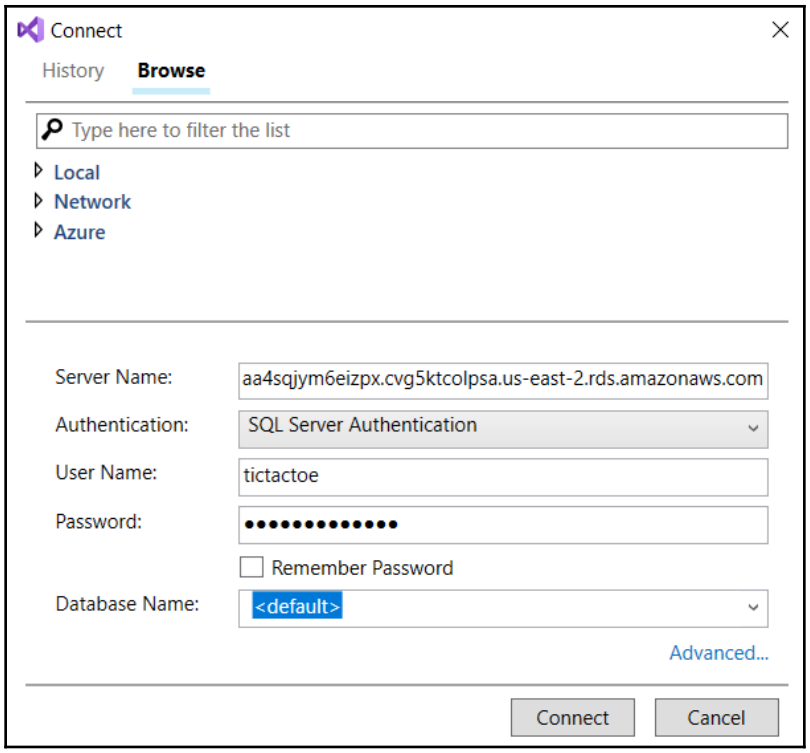
WOW! eBook
www.wowebook.org

10. Click on the **Add Rule** button, choose **MS SQL** as the **Type**, **Custom** as the **Source**, and enter a default security group, then click on the **Save** button, as shown in the following screenshot:

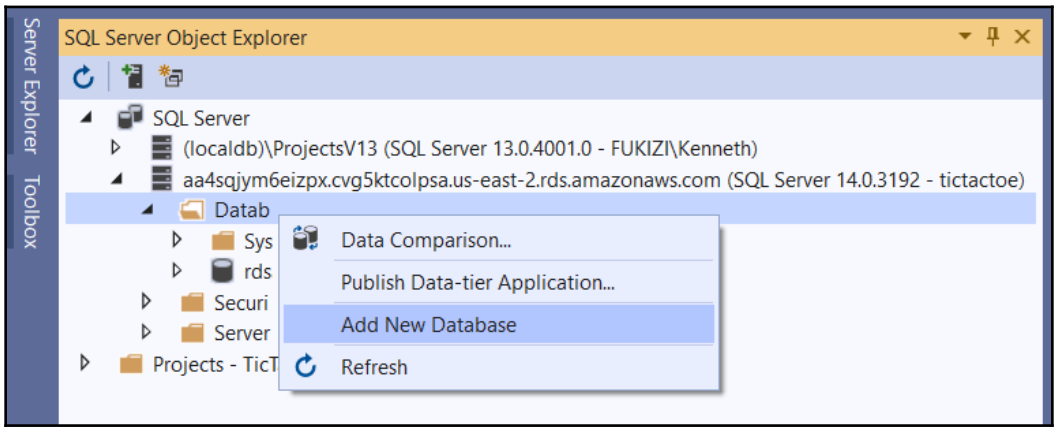
Note that you should configure stricter security group inbound rules in a real production environment, and set real IP restrictions. The source **Anywhere** should not be used for production environments.

11. You have an option to use **SQL Server Management Studio** (**SSMS**), SQL Server Object Explorer in Visual Studio 2019, or Azure Data Studio, described in a following note, to work on databases in the cloud. Let's use **SQL Server Object Explorer** in Visual Studio 2019. Open it and right-click on **SQL Server** and then **Add SQL Server**, and sign in using the endpoint address, username, and password from before, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



12. Create a new database called `TicTacToe`, as follows:



WOW! eBook
www.wowebook.org

13. Update `DatabaseConnectionString` in the `appsettings.json` file, and replace the parameters with the following corresponding values. You might recall that in *Step 7*, we mentioned the retrieved endpoint that was going to be used to update the application connection string before deployment. This is where we need to update it:

```
"Server=<YourEndPoint>;Database=TicTacToe;
  MultipleActiveResultSets=true;
  User id=<YourUser>;pwd=<YourPassword>"
```

You have successfully configured the technical environment, which means that you are now able to publish the database schema, as well as deploy the web application.

Note that Azure Data Studio is another great cross-platform option for working with SQL Server in the cloud. This is useful when you are not in need of Visual Studio for running **Entity Framework** (**EF**) Migrations. A comparison of its features compared to the commonly used SSMS, to help you decide when it is best to use Azure Data Studio or SSMS can be found here: `https://docs.microsoft.com/en-us/sql/azure-data-studio/what-is?view=sql-server-ver15`.

Are you eagerly waiting to run the application in the cloud? Just stay focused and continue a little bit further, and you will see your application running in AWS very soon.

You have three choices when it comes to publishing the database schema, as follows:

- Generate a SQL script to create the database from within Visual Studio 2019 via EF Migrations.
- Change the default connection string in `Data\GameDbContextFactory.cs` and execute the `Update-Database` instruction within the Package Manager Console.
- Run the application to create the database.

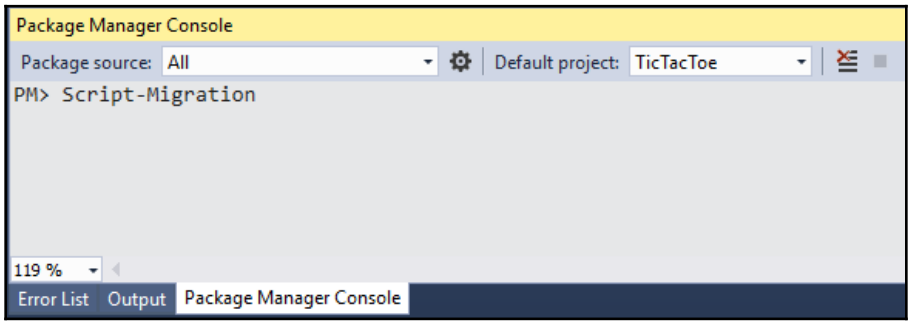
The most appropriate solution depends on the type and size of your application and its database. As a rule of thumb, it is better to generate a script and then create the database for larger applications, while for smaller applications, it is acceptable to create the database automatically when the application is running for the first time.

WOW! eBook
www.wowebook.org

Getting the application running on AWS

Let's see what needs to be done before you can see the Tic-Tac-Toe application running in AWS, as follows:

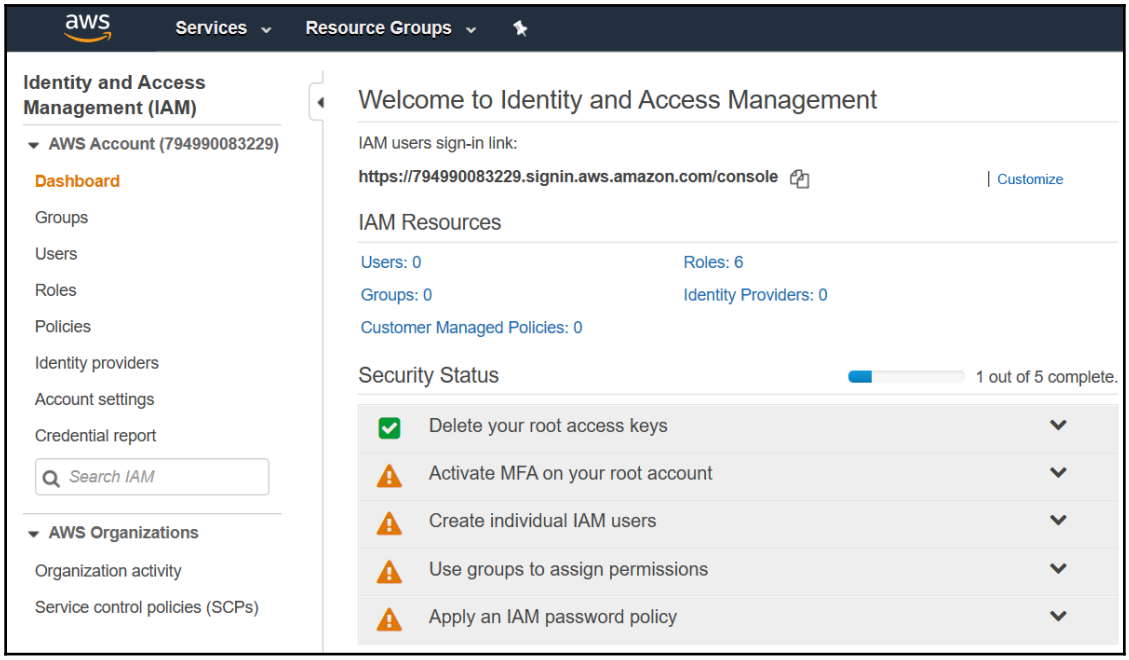
1. Open the **Package Manager Console** in Visual Studio 2019 and execute the `Script-Migration` instruction, as shown here:



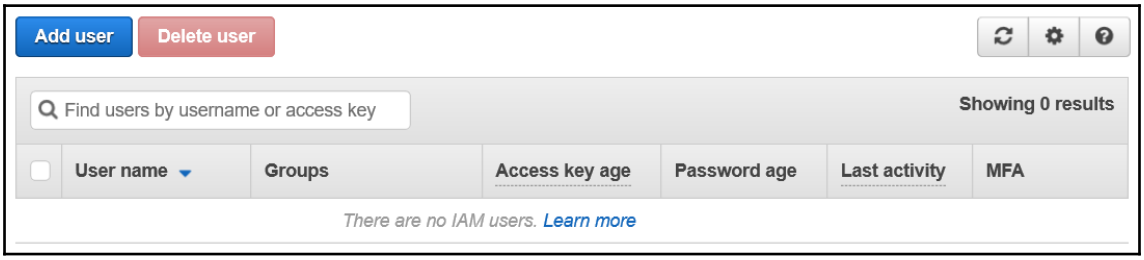
2. Take the generated script and copy it into a query window for the Amazon RDS database, then execute the script to create the database and the various database objects.
3. Download and install the **AWS Toolkit for Visual Studio 2017 and 2019** from `https://marketplace.visualstudio.com/items?itemName=AmazonWebServices.AWSToolkitforVisualStudio2017`, as shown in the following screenshot (if you are using Visual Studio Code, you can also get an AWS Toolkit for Visual Studio Code from `https://aws.amazon.com/visualstudiocode/`):

WOW! eBook
www.wowebook.org

4. Go to the **AWS Management Console**, enter `IAM` in the **AWS services** textbox, and click on the displayed link; you will be redirected to the Amazon **Identity and Access Management (IAM)** page, as shown in the following screenshot:

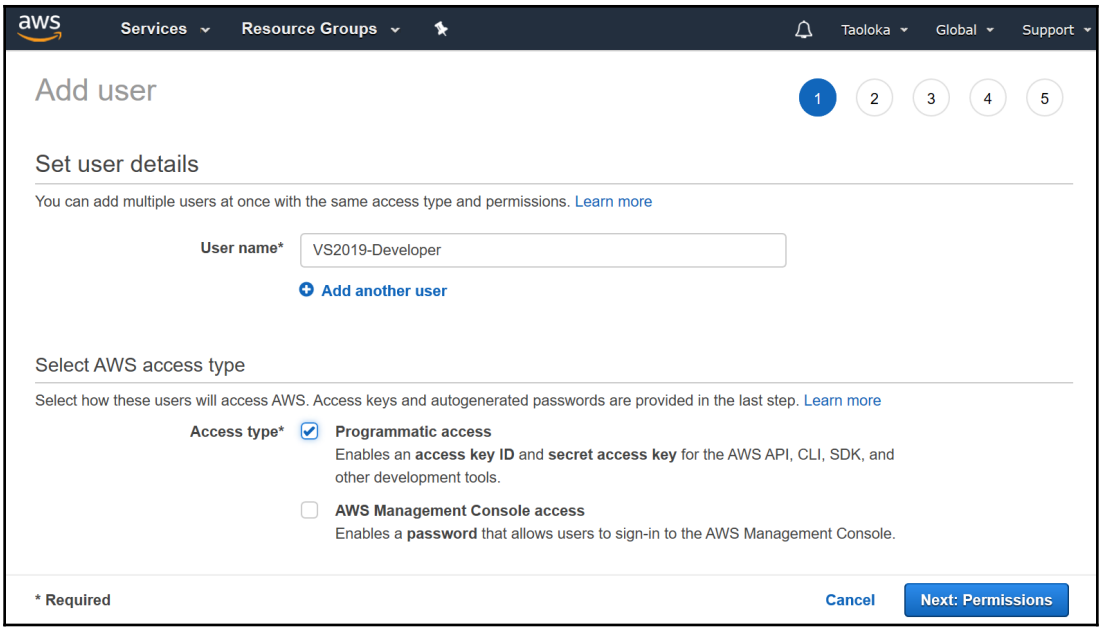


5. On the Amazon **Identity and Access Management (IAM)** page, click on **Create individual IAM Users** and then on **Manage Users** and click the **Add user** button, as shown in the following screenshot:

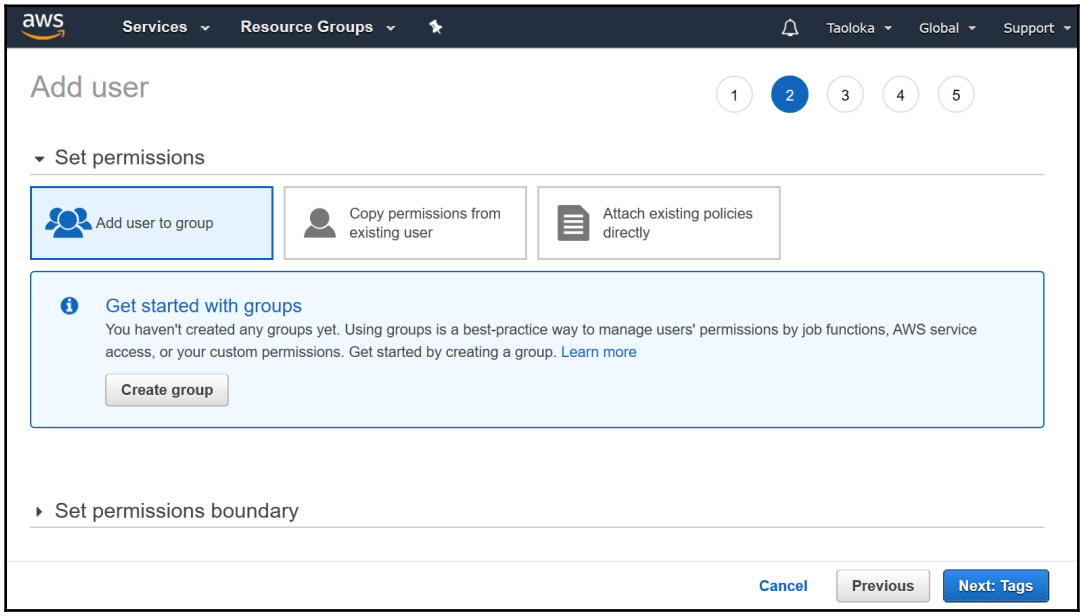


6. On the **Add user** page, give the new user a meaningful username and grant them **Programmatic access**, then click on the **Next: Permissions** button at the bottom of the page, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

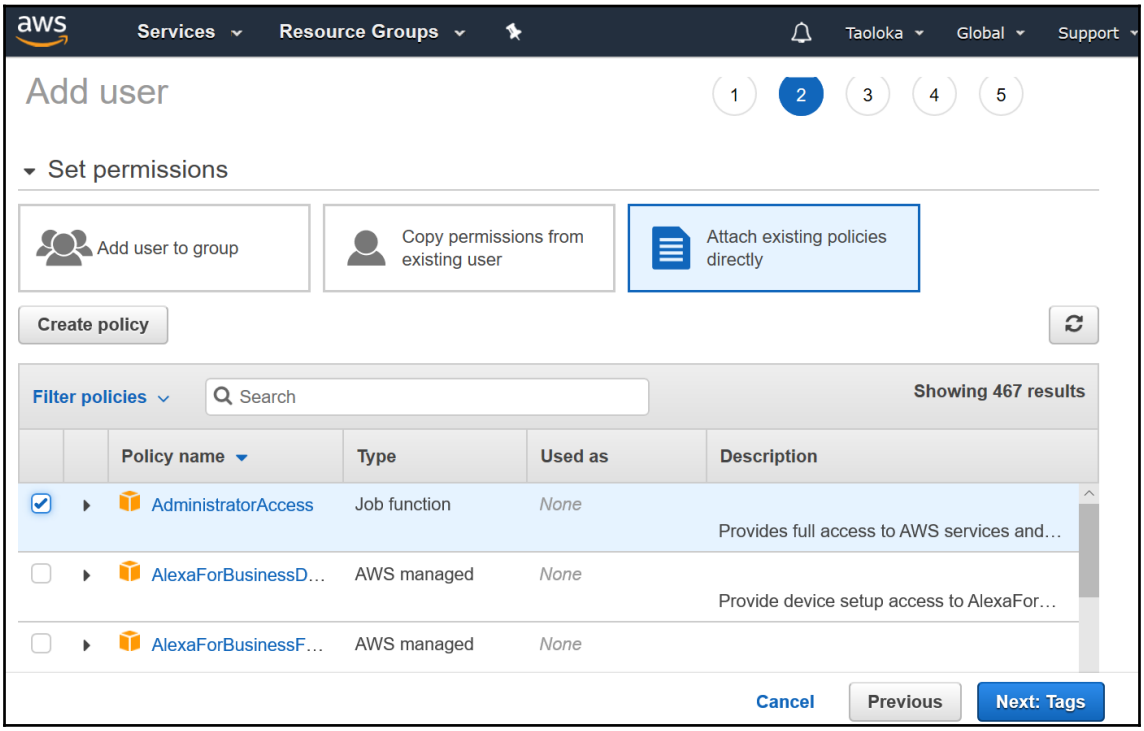


7. You now have to set the permissions for the new user; for that, click on the **Attach existing policies directly** button, as shown in the following screenshot:



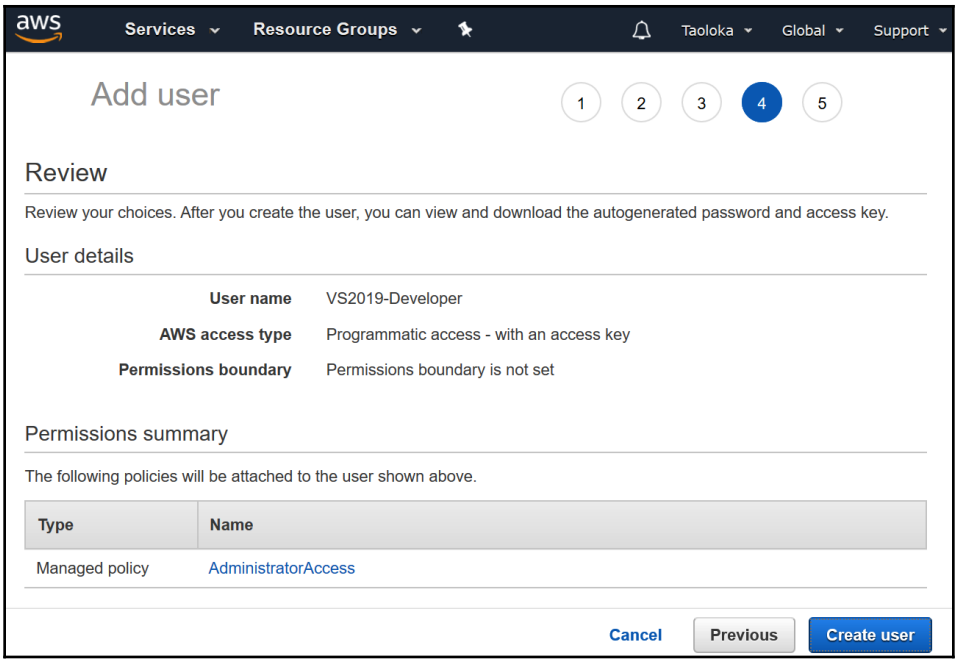
WOW! eBook
www.wowebook.org

8. Select **AdministratorAccess** from the existing policies and click on the **Next: Tags** button at the bottom of the page, as shown in the following screenshot:

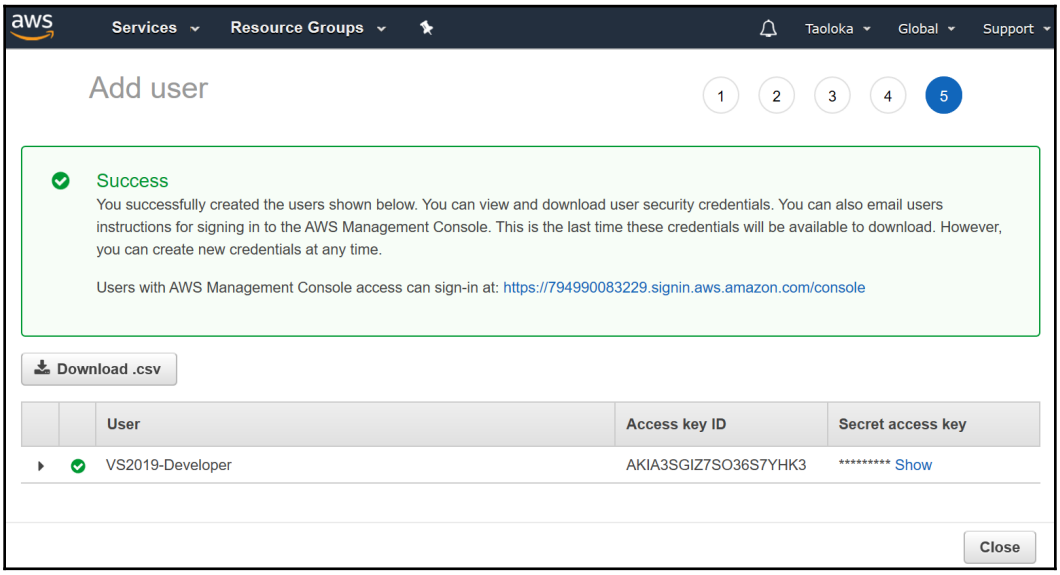


9. We can ignore tags for now, since we don't intend to have many users, and just go to **Next: Review** and verify that the **User name** and **AWS access type**, as well as the selected policies, are correct, then click on the **Create user** button, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

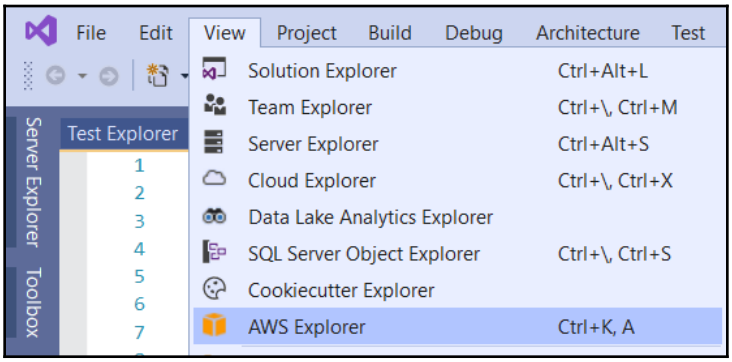


10. Wait for the new user to be created; when the success page is displayed, you can then download the `.csv` file, which we will use to configure Visual Studio 2019 with AWS, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

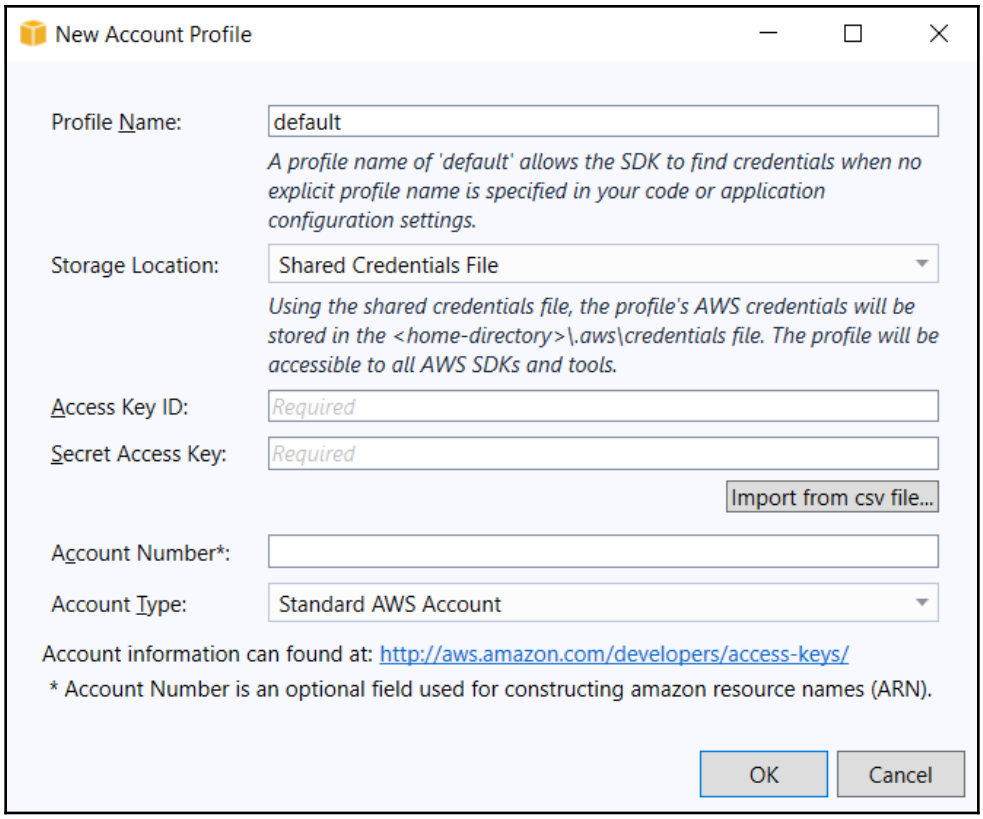
11. Open Visual Studio 2019 and display **AWS Explorer** by going to **View | AWS Explorer**, as shown in the following screenshot:



12. Click on the **New account profile** button (the only active button), as shown in the following screenshot:

13. A wizard will be displayed; leave the **Profile Name** as **default** and fill in the **Access Key ID** and **Secret Access Key** fields with the values coming from the `.csv` file you have downloaded before, during the new user creation process on AWS, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



14. Since AWS is based on IIS as the host for .NET Core applications, you now have to add a `web.config` file to the `TicTacToe` application, which specifies the IIS web server properties. The handlers, which are meant to process external requests and give a response, are assigned `aspNetCore` attributes and are set to allow all HTTP verbs such as `GET`, `POST`, `PUT`, `DELETE`, and many others. The processing path for the `aspNetCore` instance is assigned with the our `dll` path, and we also make sure that Windows authentication is supported, by setting the `forwardWindowsAuthToken` attribute to `true`, as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*"
        modules="AspNetCoreModule"
         resourceType="Unspecified" />
```


WOW! eBook
www.wowebook.org

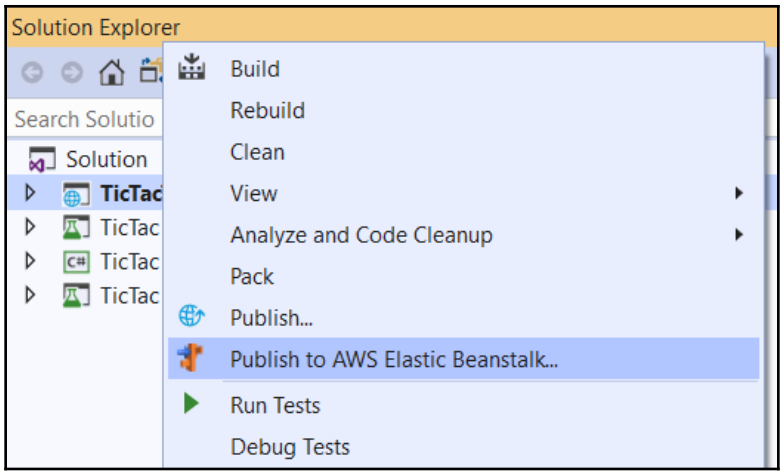
```
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\TicTacToe.dll"
      stdoutLogEnabled="true"
      stdoutLogFile=".\logs\stdout"
      forwardWindowsAuthToken="true" />
  </system.webServer>
</configuration>
```

15. Furthermore, we have to enable IIS integration. For us to do that, we open the `Program.cs` file and change the WebHost builder configuration to enable IIS integration, by adding `webBuilder.UseIISIntegration()`, as follows:

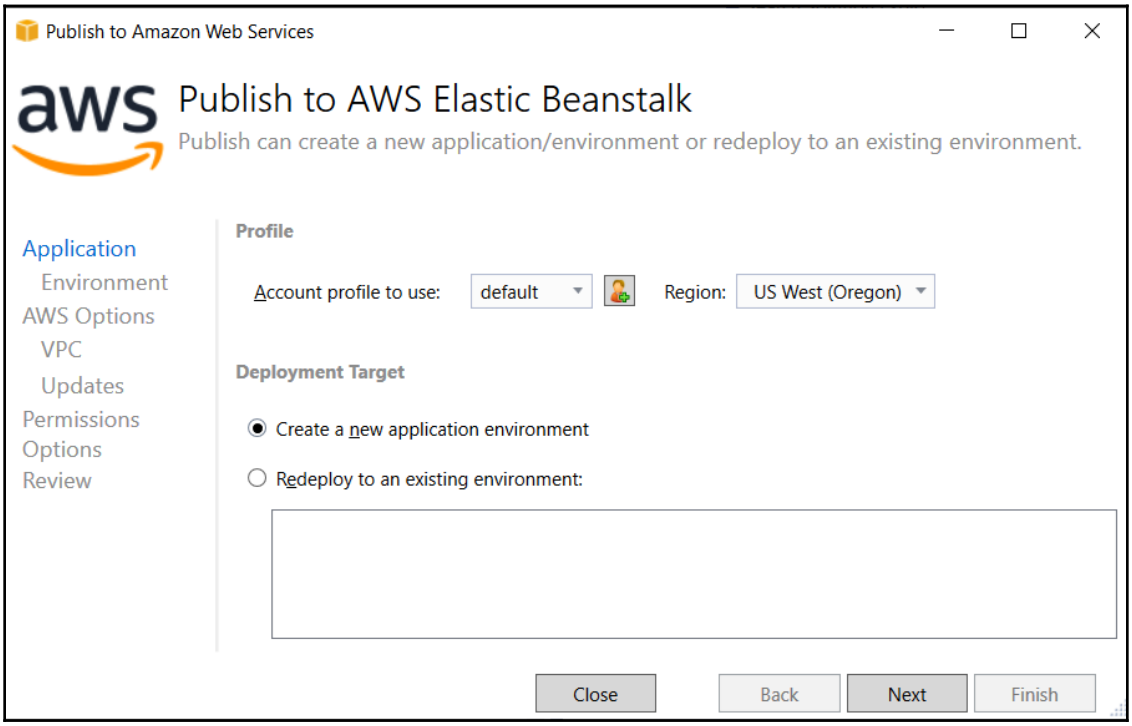
```
public static IHostBuilder CreateHostBuilder(string[]
 args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.CaptureStartupErrors(true);
            webBuilder.PreferHostingUrls(true);
            webBuilder.UseUrls("http://localhost:5000");
            webBuilder.ConfigureLogging((hostingcontext,
              logging) =>
            {
                logging.AddLoggingConfiguration(
                 hostingcontext.Configuration);
            });
            webBuilder.UseIISIntegration();
        });
```

16. Right-click on the **TicTacToe** project and click on **Publish to AWS Elastic Beanstalk...** in the context menu, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

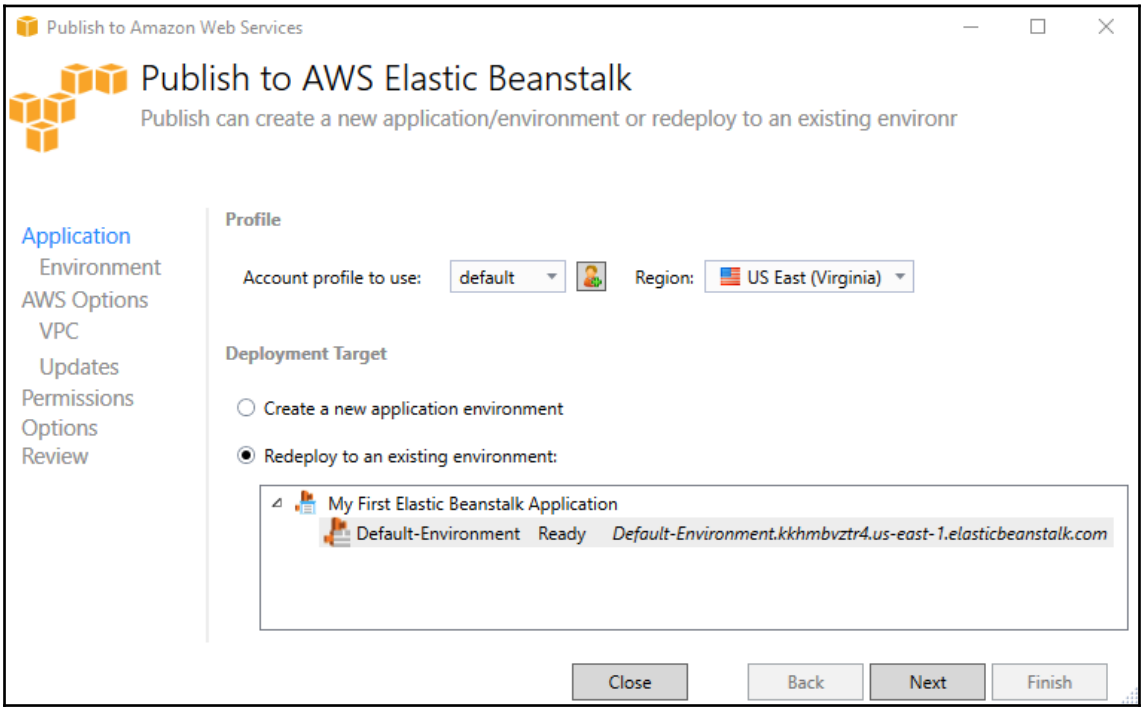


17. A wizard will be displayed; click on **Create a new application environment** and click on the **Next** button, as shown in the following screenshot:



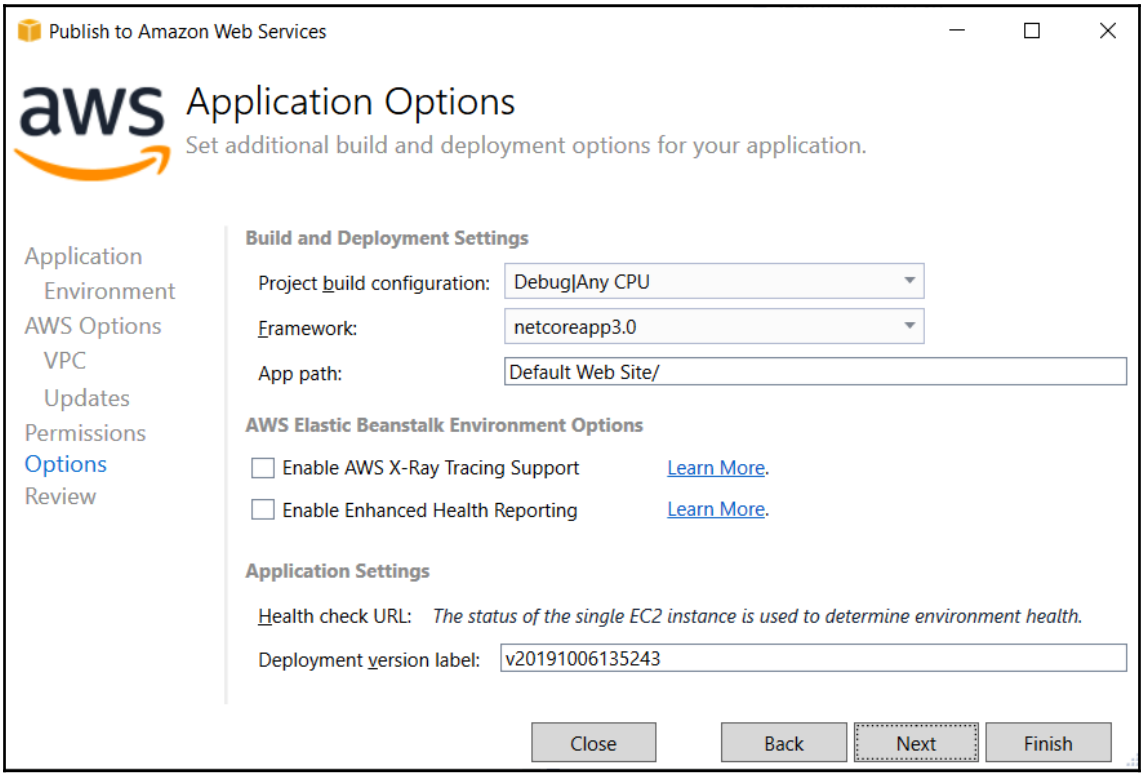
WOW! eBook
www.wowebook.org

18. You have three options for the environment—whether dev, test, or production—but select the default environment you have created before, then click on the **Next** button, as shown in the following screenshot:



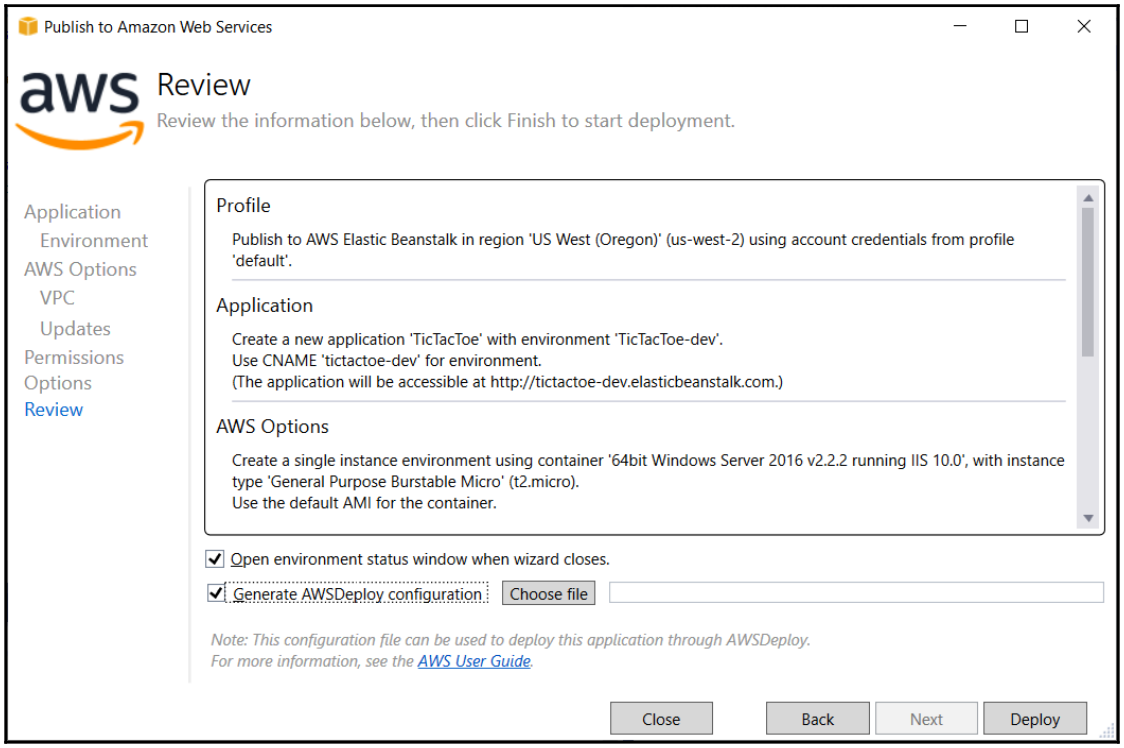
WOW! eBook
www.wowebook.org

19. Verify that the **Framework** version is set to **netcoreapp3.0**, signifying an ASP.NET Core 3 application, and leave all default values, then click on the **Next** button, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

20. Select **Generate AWSDeploy configuration**, which will allow you to redeploy a copy of your application with AWS, then click on the **Deploy** button, as shown in the following screenshot:

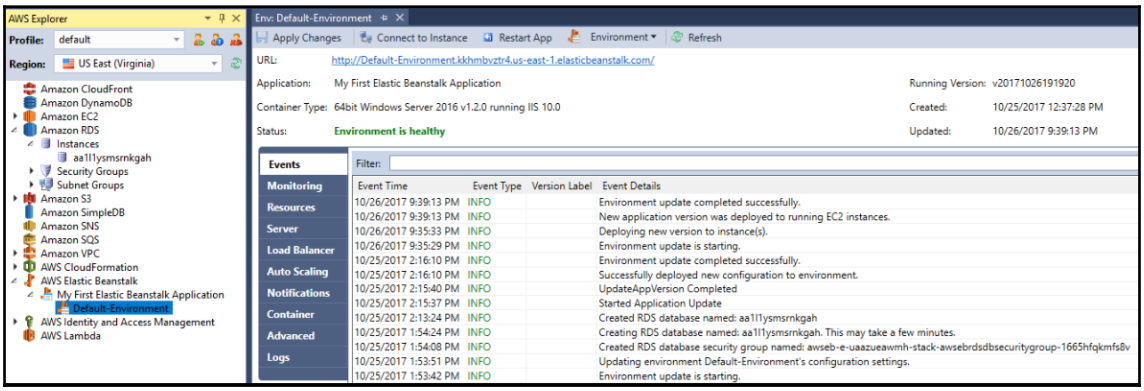


21. The deployment will start; you can see the advancement of the deployment process by going to **Output | Amazon Web Services**, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

```
Output
Show output from: Amazon Web Services
 ......uploaded 77% of application deployment package
 ......uploaded 81% of application deployment package
 ......uploaded 85% of application deployment package
 ......uploaded 88% of application deployment package
 ......uploaded 92% of application deployment package
 ......uploaded 96% of application deployment package
 ......uploaded 99% of application deployment package
 ......uploaded 100% of application deployment package
 ....upload complete
 ....creating new version 'v20171026191920' for application 'My First Elastic Beanstalk Application
 ...requesting update of environment 'Default-Environment' with application version 'v20171026191920'
 Publish to AWS Elastic Beanstalk environment 'Default-Environment' completed successfully

Package Manager Console  Error List  Output
```

22. When the application is deployed, you can use AWS Explorer to get the URL of the application, as follows:



23. Open a browser and go to the application URL in AWS, start the application, and try to register a new user.

Note that if the application is not working as expected, you will get a `404 Not Found` HTTP response. Everything is working locally and the deployment in AWS was successful, but something is wrong. You will see in the next chapter, which is about logging and monitoring, how to analyze, diagnose, understand, and fix this problem.

WOW! eBook
www.wowebook.org

Congratulations—you have successfully deployed your first application in the public cloud. It is now available to the outside world, and users can connect to it and start working with it.

This concludes the examples in relation to AWS. However, we still have some compelling content, since we will explore how to deploy to other targets, such as Microsoft Azure and Docker containers, in the next sections; so, stay sharp, and continue reading the following sections.

Deploying applications in Microsoft Azure

Microsoft Azure is a public cloud computing platform provided by Microsoft for building, testing, deploying, and managing applications and services within globally available Microsoft data centers all around the world. Microsoft Azure is not only meant to cater for tooling that comes from Microsoft, in terms of programming languages and frameworks, but also includes third-party languages, frameworks, and tools, both proprietary or open source.

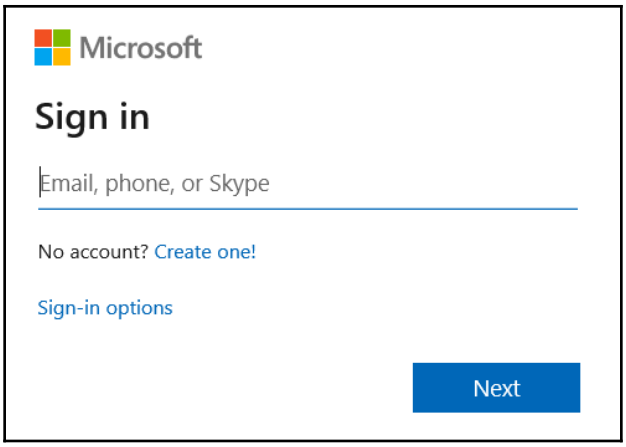
When deploying web applications in Microsoft Azure, you basically have four choices, as follows:

- Azure App Service
- Azure Service Fabric
- Azure Container Service
- Azure Virtual Machines

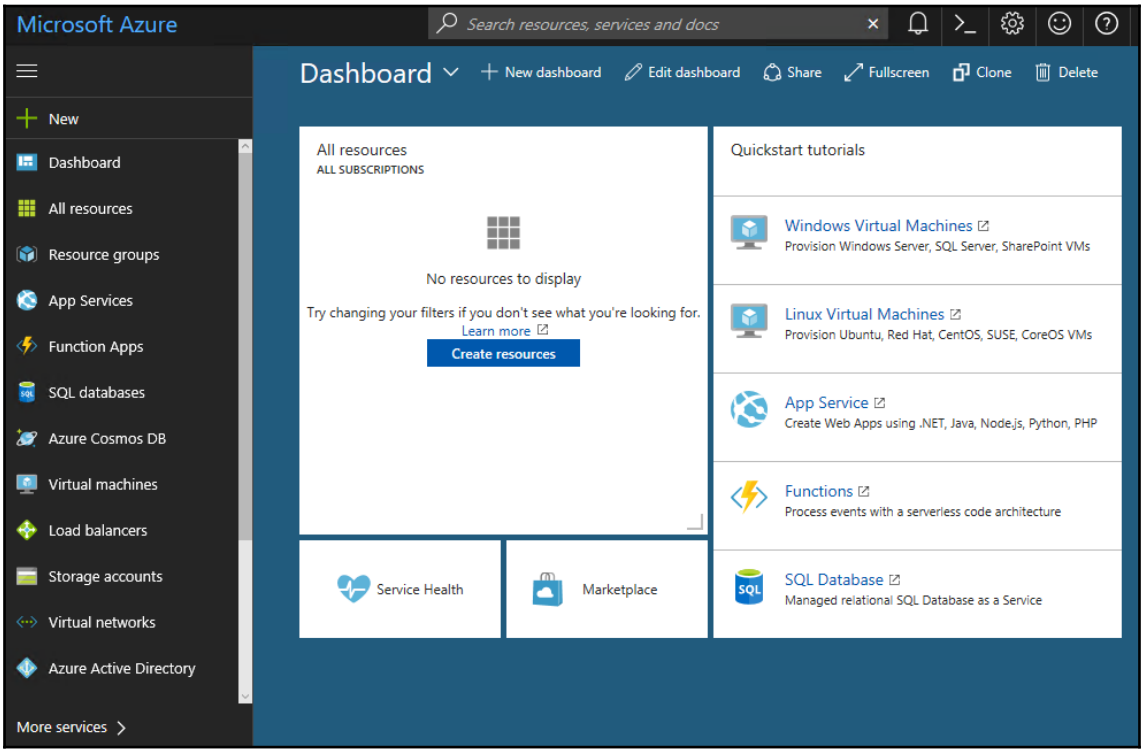
However, before you can start deploying your applications in Microsoft Azure, you need to sign up for a subscription; so, let's do that right now, as follows:

1. You need a Microsoft account to be able to sign up for a Microsoft Azure subscription. You can use the same one you have used for your **Azure DevOps** subscription, but if you do not have one yet, create it by going to `http://www.live.com` and clicking on the **Create one!** link, as shown in the following screenshot:

WOW! eBook
www.wowebook.org

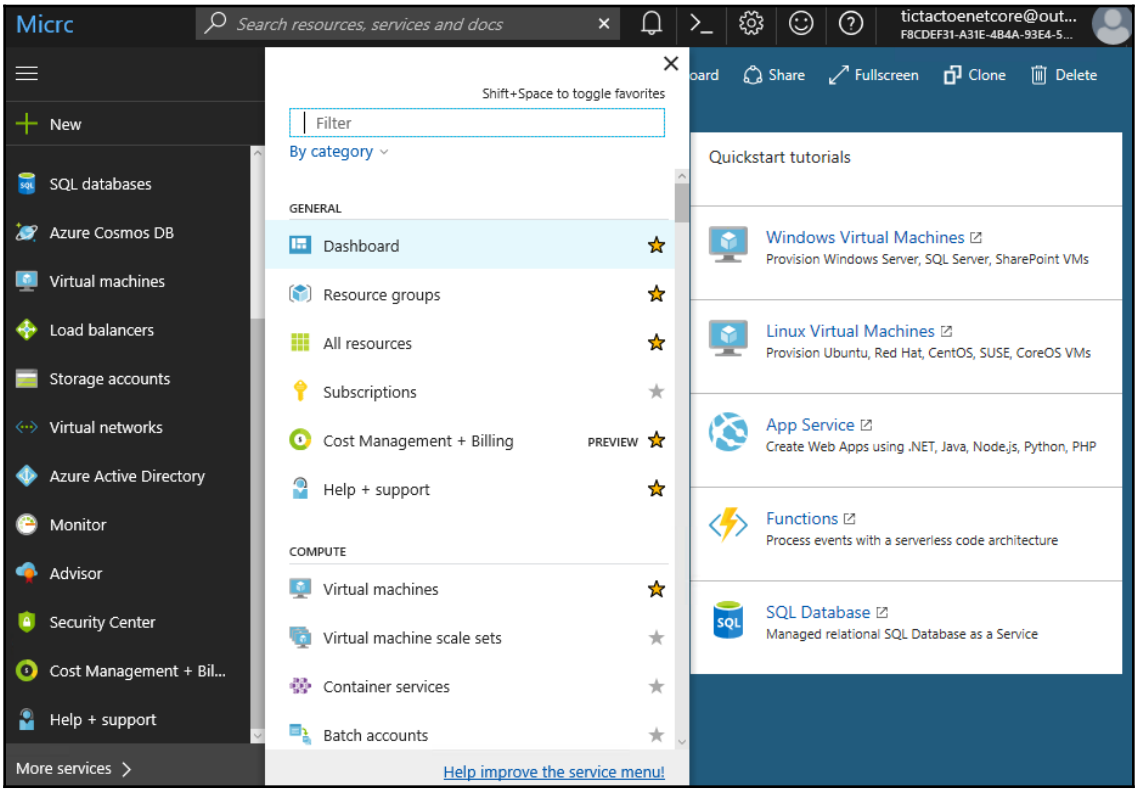


2. Go to `https://portal.azure.com` and log in with your Microsoft account; you will be asked if you want to take a tour. Select **Maybe later** (you should really take the tour later, though!), and you will be redirected to the Microsoft Azure management portal, as shown in the following screenshot:



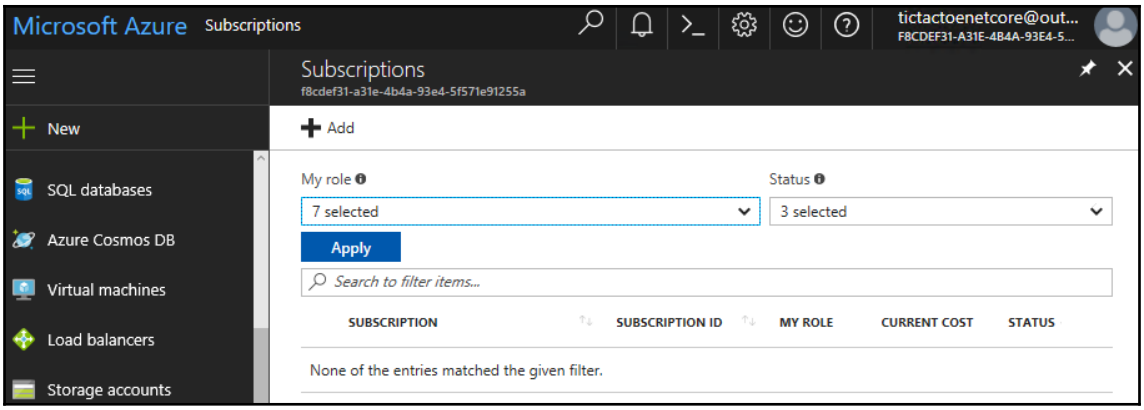
WOW! eBook
www.wowebook.org

3. Click on **More services** at the bottom of the left-hand menu, then click on the **Subscriptions** button, as shown in the following screenshot:

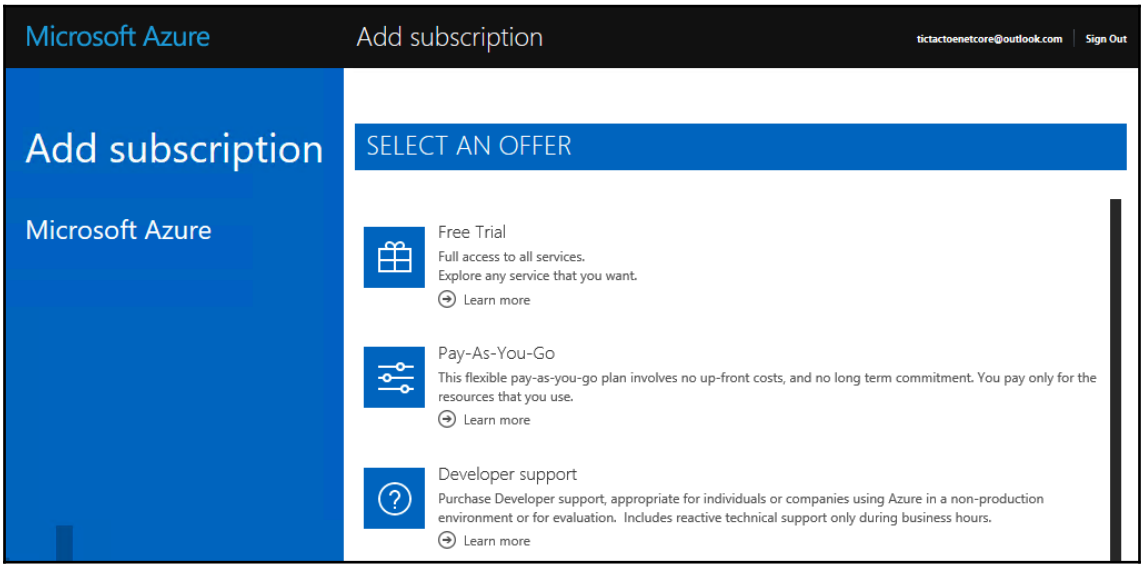


4. Click on the **Add** button, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



5. Click on the **Free Trial** button and fill in the different forms until you have created your Microsoft Azure subscription, as follows:



Note that there is no credit card required for Microsoft Azure (unlike AWS), and this is great, especially if you are a student.

Exciting! You are now ready to provision the technical environment and, then, deploy your ASP.NET Core 3 web applications to the Microsoft Azure data centers all around the world!

WOW! eBook
www.wowebook.org

Deploying applications in Microsoft Azure App Service

Azure App Service is a PaaS offering for web-based applications in Microsoft Azure that which includes Auto Scaling. In this regard, it is comparable to AWS Elastic Beanstalk, which you have already seen before, in the section on AWS.

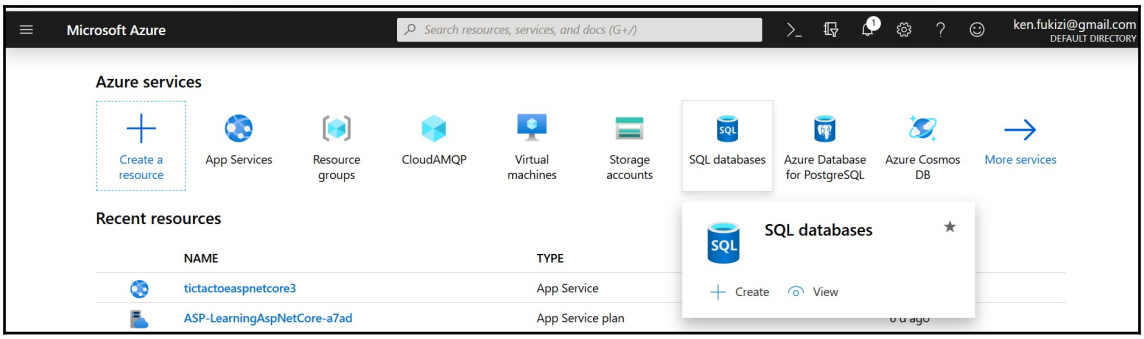
Azure App Service removes the need for a managing infrastructure; instead, you only need to be concerned about building and hosting your applications. For a full DevOps approach, it is advisable to use this PaaS service, if you want to work with Microsoft Azure.

For more information on Microsoft Azure App Service, check out `https://docs.microsoft.com/en-us/azure/app-service/app-service-web-overview`.

Getting an Azure App Service instance running

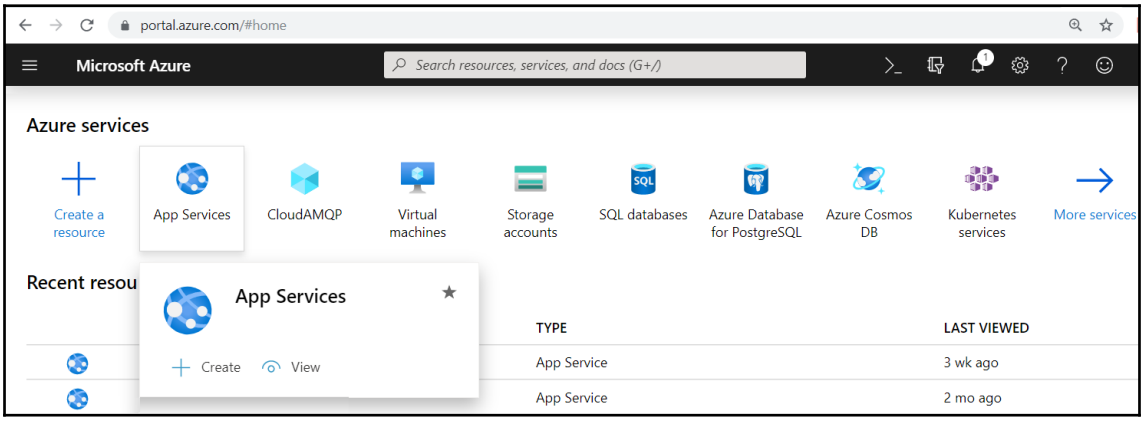
The following examples illustrate how we can prepare to deploy the Tic-Tac-Toe application to Azure App Service step by step:

1. Go to the Microsoft Azure management portal, and you will find there are many services available for you to use, including the most commonly used ones that are placed prominently on the welcome page, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

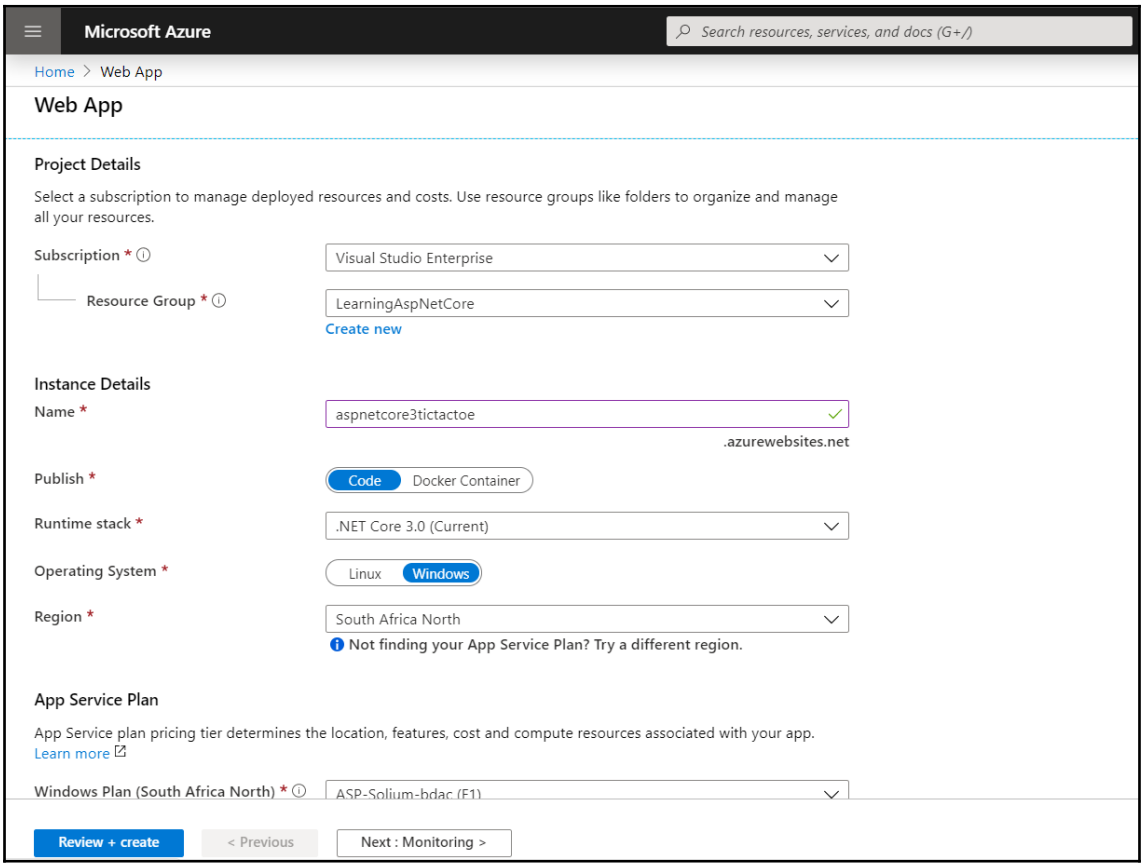
2. **App Services** is the one we need at this point, so hover with your mouse pointer over the **App Services** icon, and it will show the following popup. Click on **Create**, as follows:



3. You will get a new **Web App** form. Fill in the **Project Details**. In this screenshot, I already have a **Visual Studio Enterprise** subscription, but you can use the trial subscription you created or, indeed, any other subscription you may have. I already have a **Resource Group** created, but as you can see, you can create a new **Resource Group**; and if you hover on the information icon, it will tell you that a **Resource Group** is a simple collection of resources that share the same life cycle, permissions, and policies. Fill in a unique name for your application, and choose to publish the application as **Code**. Choose **.NET Core 3.0 (Current)** as a runtime stack, and the operating system will automatically be selected for you. For the region, I have chosen **South Africa North** as it is the one closest to me, but you can choose whatever region you feel is closer to your targeted audience in a real-life app.

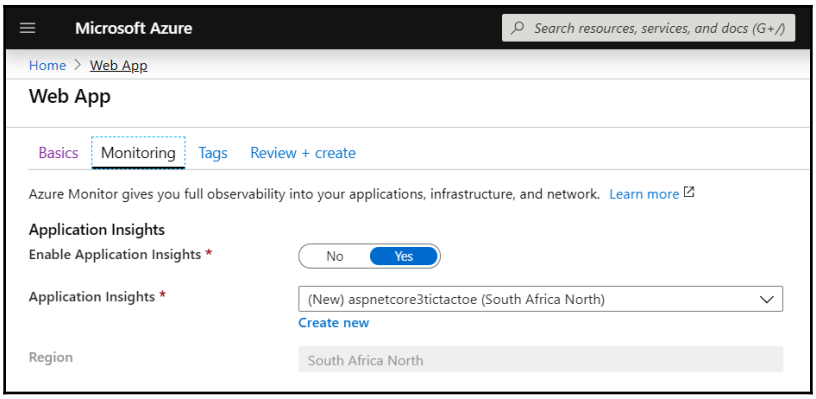
WOW! eBook
www.wowebook.org

If you don't have an App Service Plan that decides what you will be using and paying for, please create it, and be sure to double-check that you have selected a free tier that uses shared infrastructure and allows for up to 1 GB memory, as follows:

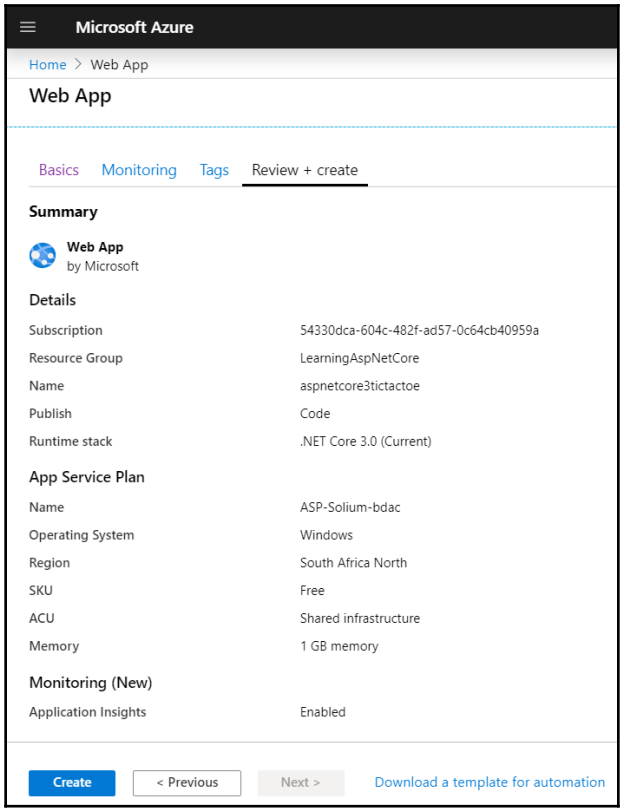


4. After filling in the preceding form and clicking on **Next: Monitoring**, you will get the following screen, where you can choose if you want to enable **Application Insights**. Select **Yes**, and then click **Next: Tags**, as follows:

WOW! eBook
www.wowebook.org

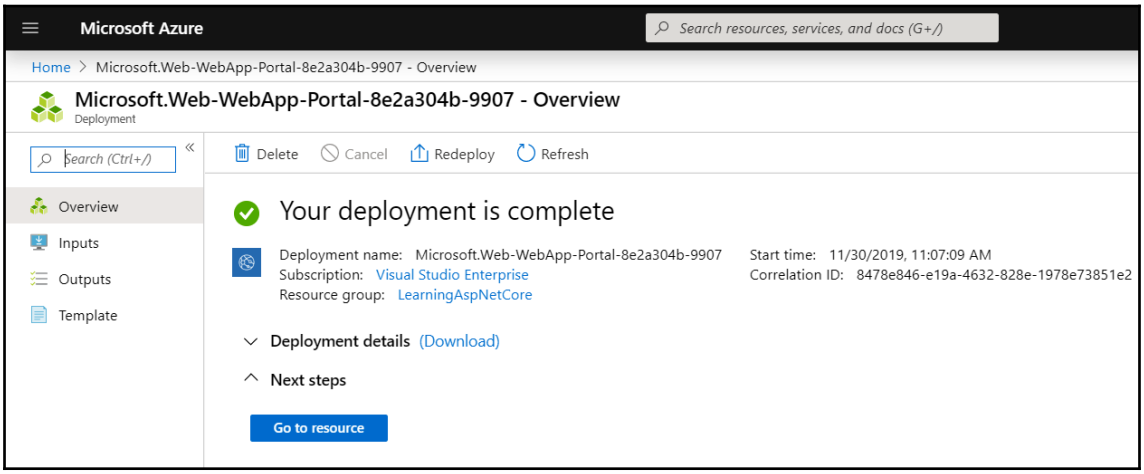


5. After clicking **Next: Tags**, you will get to the **Tags** section. We don't need them for this application, so click on the last **Next** button to review and create, where you will be presented with the following summary:



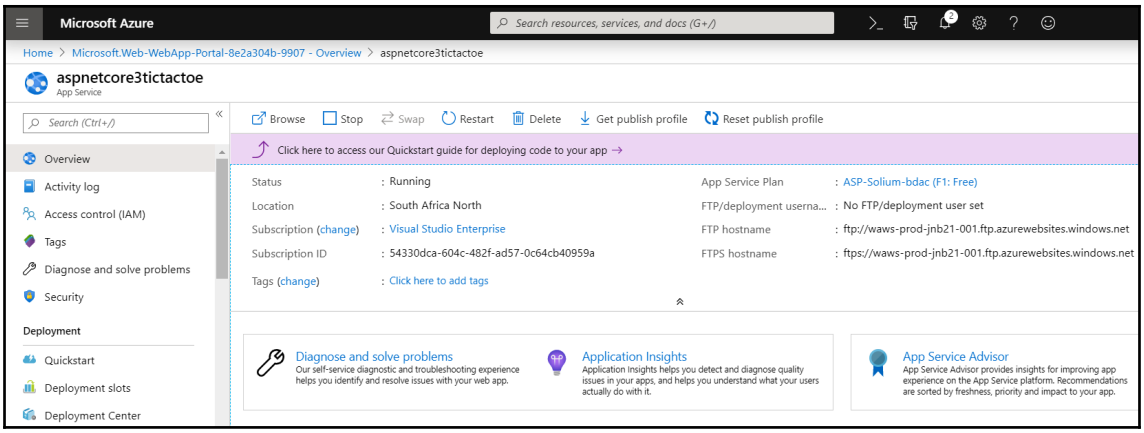
WOW! eBook
www.wowebook.org

6. Click on the **Download a template for automation** link, and this will download a `template.zip` folder, with two files inside: `parameters.json` and `template.json`, which we may need for future use. After the download, go back to the web app through the navigation link, and it will still have the summary. Click on **Create**, and you will get a deployment underway screen and then a deployment complete screen, as follows:

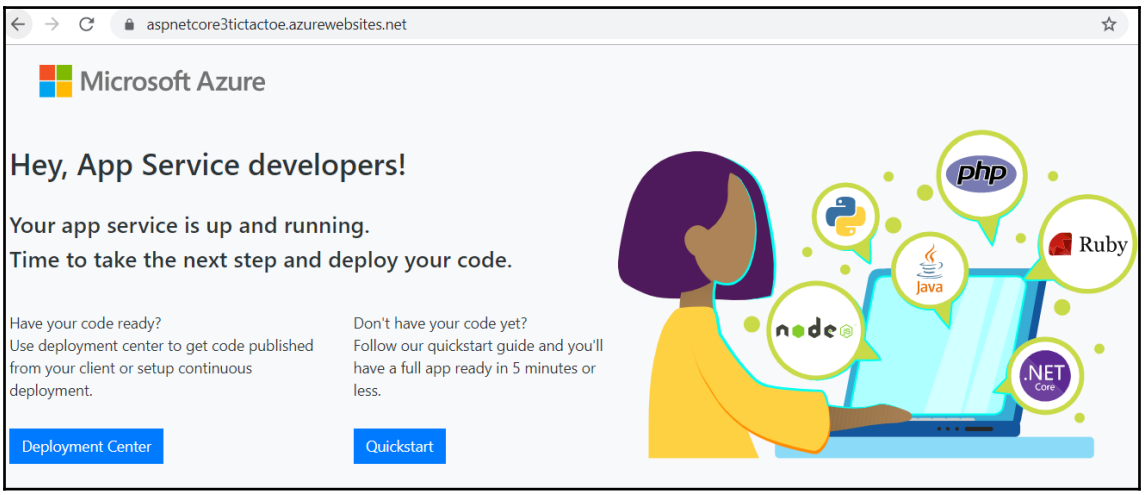


7. Click on **Go to resource** from the preceding screen, and you will then be presented with a dashboard for your deployed application, as follows:

WOW! eBook
www.wowebook.org



8. At this point, if you click on the link with the unique name you created in *step 3*: `https://[your-unique-project-name].azurewebsites.net/`, you will see that you already have a website, as shown in the following screenshot:

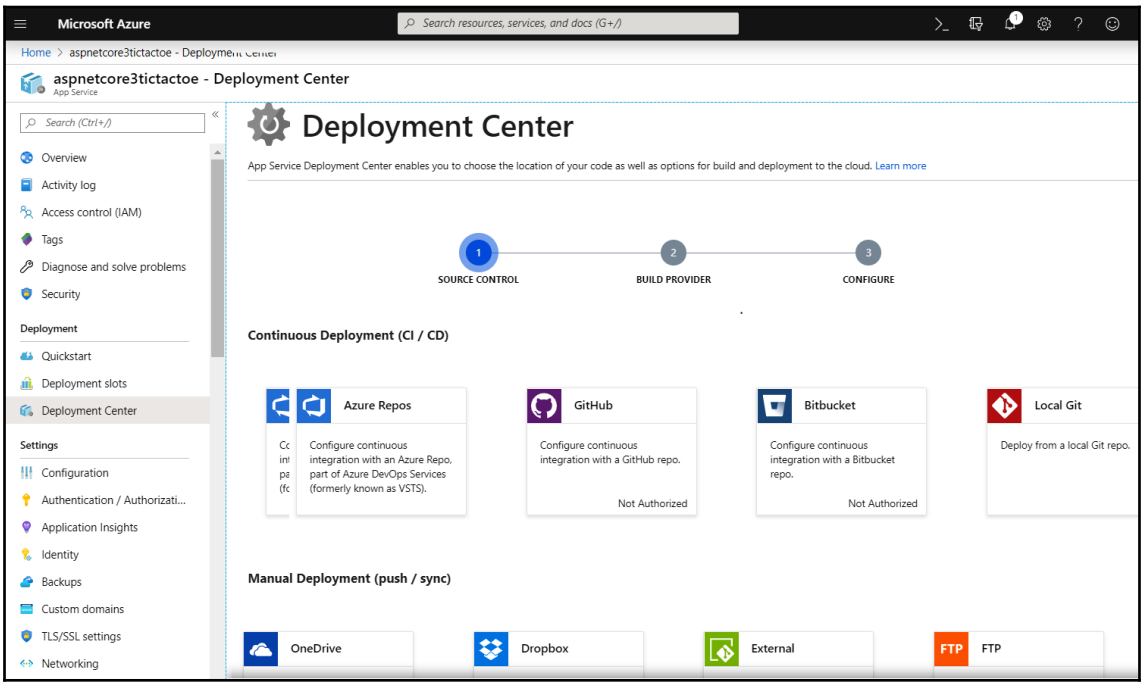


Congratulations! If you have reached this far, you have a healthy app service running on Azure, but as you will notice, we still don't have our Tic-Tac-Toe demo application running on Azure. Let's see how we can do that next, in the following section.

WOW! eBook
www.wowebook.org

Publishing your code on Azure

Now that you know you have your App Service instance running, go ahead and click on the **Deployment Center** button shown in the preceding screenshot, and you will get the screen shown in the following screenshot. You can also go to **Deployment Center** by going to the home page, browsing through recent resources, and selecting your project, then select **Deployment Center** in the left pane, under **Deployment**, as follows:



You will be glad to see that you have several options through which you can deploy your application. You will remember from `Chapter 3`, *Continuous Integration Pipeline in Azure DevOps* that we did have our application in **Azure Repos**, and you might have been updating your code throughout the chapter on Azure Repos, or a local Git project. Either way, we have an option to deploy from **Azure Repos** or Local Git available to us.

GitHub now has a cool feature called GitHub Actions that makes deployment much easier. It has a lot of workflows provided by its vibrant community, such as *Deploy to Kubernetes*, *AWS*, *and Azure App Service*, and you can create your own actions using the starter workflows here: `https://github.com/actions/starter-workflows`. You can find out more about GitHub Actions at `https://github.com/features/actions`.

WOW! eBook
www.wowebook.org

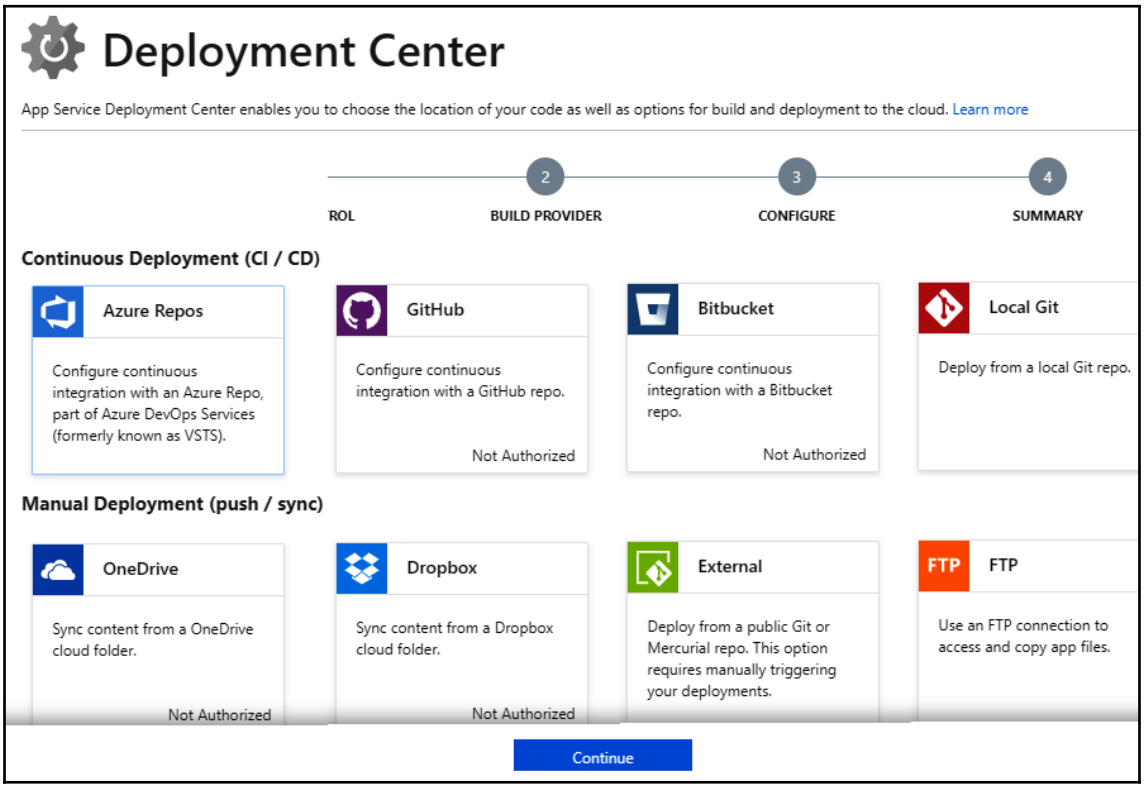
We also have an option to deploy through a **GitHub** repository or **Bitbucket**, but we will need to do some more configurations, and this is not within the scope of this book.

For demonstration purposes, we will use Azure Repos, as demonstrated in the next section.

Continuous integration with Azure Repos

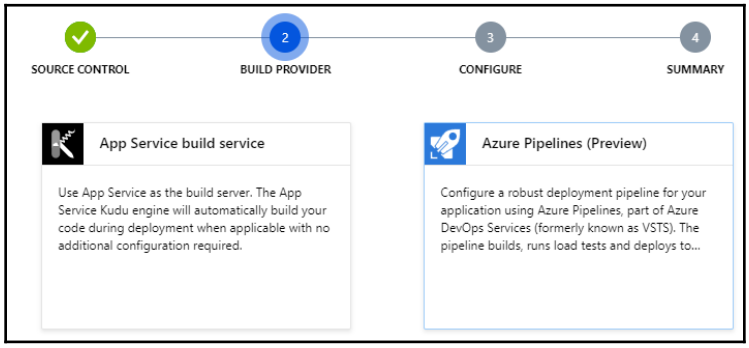
If you have been updating the code on Azure Repos, you can simply click on the **Azure Repos** button to continue the process. If you have not been updating, and have been using a local development environment, please make sure you check out your Azure Repos code from `Chapter 3`, *Continuous Integration Pipeline in Azure DevOps*, update it with the latest, and commit. Follow these steps:

1. In the **Deployment Center**, click on **Azure Repos** and **Continue**, as shown in the following screenshot:

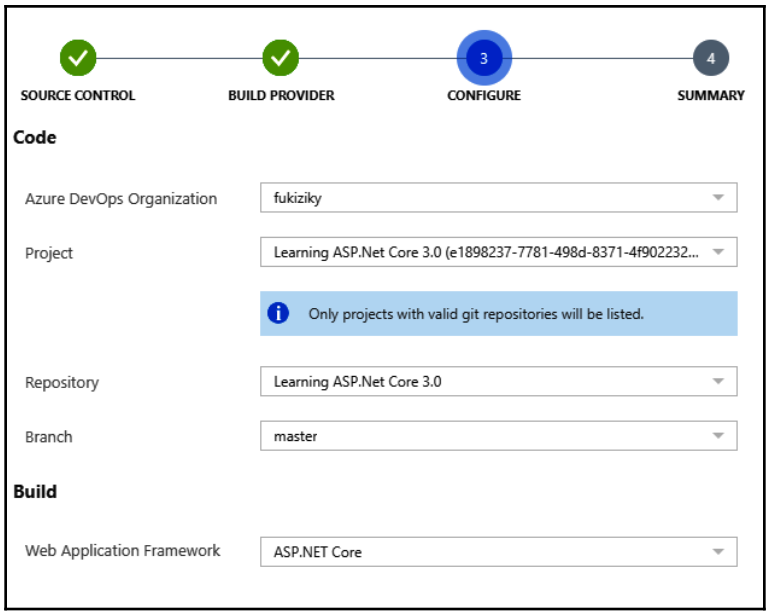


WOW! eBook
www.wowebook.org

2. You will be presented with two options for your build provider, either the **App Service build service** or the **Azure Pipelines (Preview)**. The main difference between them is that Azure Pipelines does a bit extra, such as running load tests, and deploying to staging first and then to production. At this point, make sure that you do have both a staging and a production slot, which you can add by clicking on the **Deployment** slots menu on the left. Select **Azure Pipelines (Preview)** and click **Continue**, as follows:

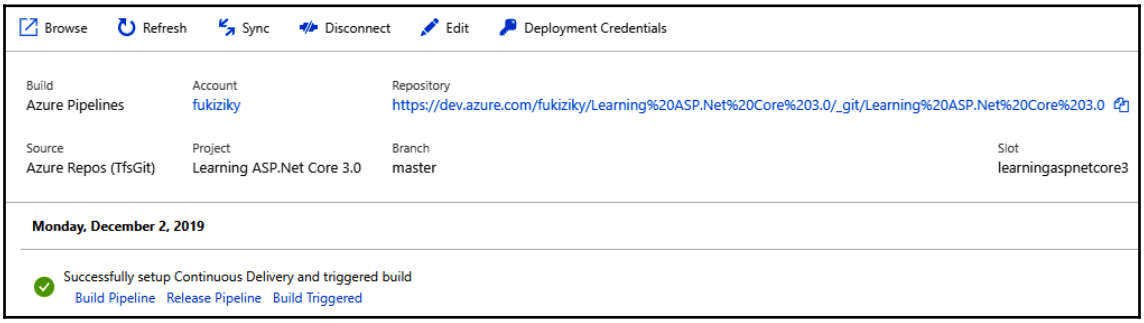


3. You then add the configurations for the Azure Repos project you created in `Chapter 3`, *Continuous Integration Pipeline in Azure DevOps*, as in the following screenshot. Click **Continue**, as follows:

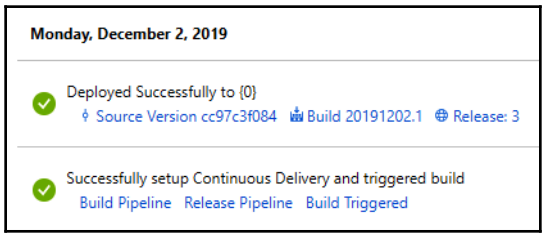


WOW! eBook
www.wowebook.org

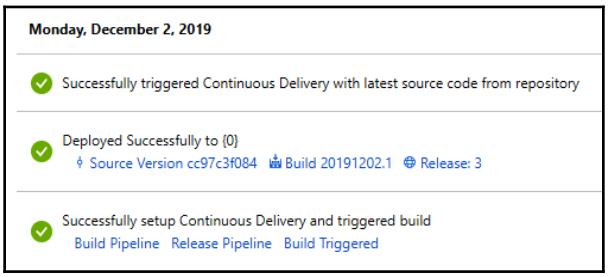
4. You are then presented with a summary page, which you can use to check if all your parameters are OK. If not, you can go back and correct; but assuming everything is correct, click **Finish**, and if everything went correctly, you should see the following **Successfully setup Continuous Delivery and triggered build** message:



5. Do a refresh, and you will see a deployed successfully message. The message will also contain the release version, and links to the build details and source version, as follows:



6. Click on **Sync** to make sure that your web app has the latest code, and you will get the message **Successfully triggered Continuous Delivery with latest source code from repository**, as shown in the following screenshot:



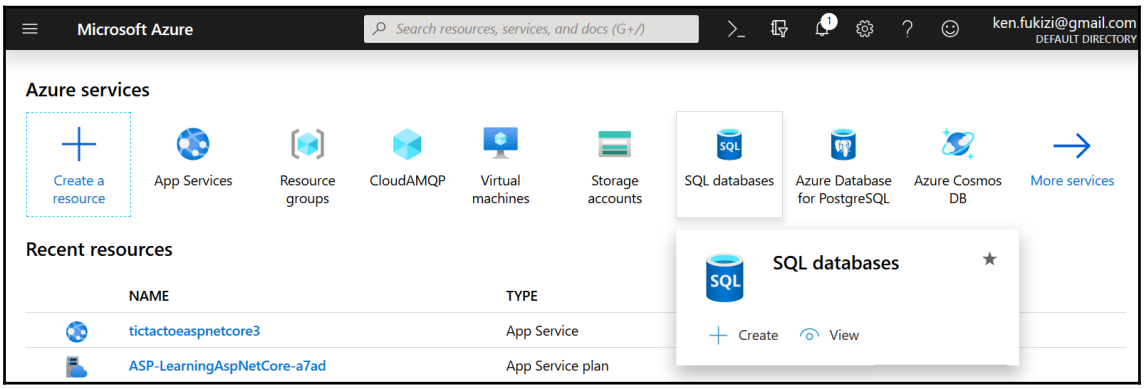
WOW! eBook
www.wowebook.org

This is good news. Your code is now on the Azure portal, but if you check the URL for your project—`https://[your-unique-project-name].azurewebsites.net/`— you will be faced with an error. Don't despair; it's still a good sign. We have our code on the site, but we have not connected to any database. That is what we are going to look at in the next section.

Connecting the database

In previous versions of Microsoft Azure, there used to be an option to have a **Web App + SQL** as a single Azure service, but of late they have been split. We will need to create a separate database instance, and the following are the steps we will need to go through to get the database up and running, and ready to be used by our demo application:

1. Go to your personal home page for `portal.azure.com` and hover on **SQL databases**, then click on the **Create** button:



WOW! eBook
www.wowebook.org

2. Fill in the required information on the resulting form, as follows:

Microsoft Azure

Home > Create SQL Database

Create SQL Database
Microsoft

Basics • Networking Additional settings Tags Review + create

Create a SQL database with your preferred configurations. Complete the Basics tab then go to Review + Create to provision with smart defaults, or visit each tab to customize. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Visual Studio Enterprise

Resource group * LearningAspNetCore
Create new

Database details

Enter required settings for this database, including picking a logical server and configuring the compute and storage resources

Database name * TicTacToe

Server * Select a server
Create new

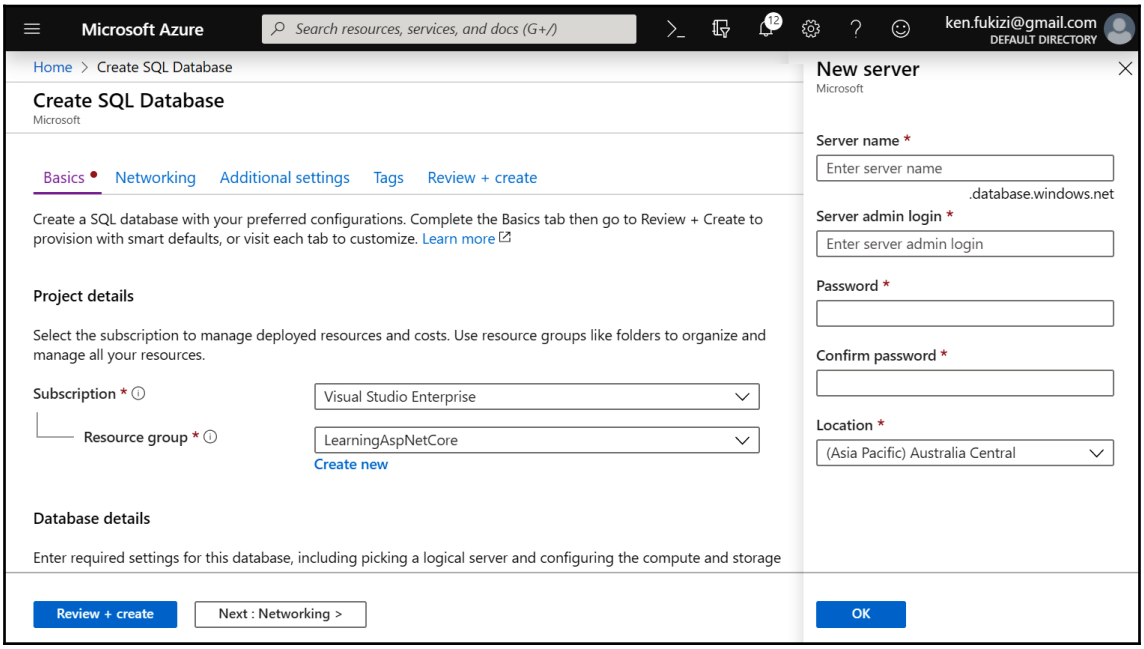
The value must not be empty.

Want to use SQL elastic pool? * Yes No

Compute + storage * Please select a server first.
Configure database

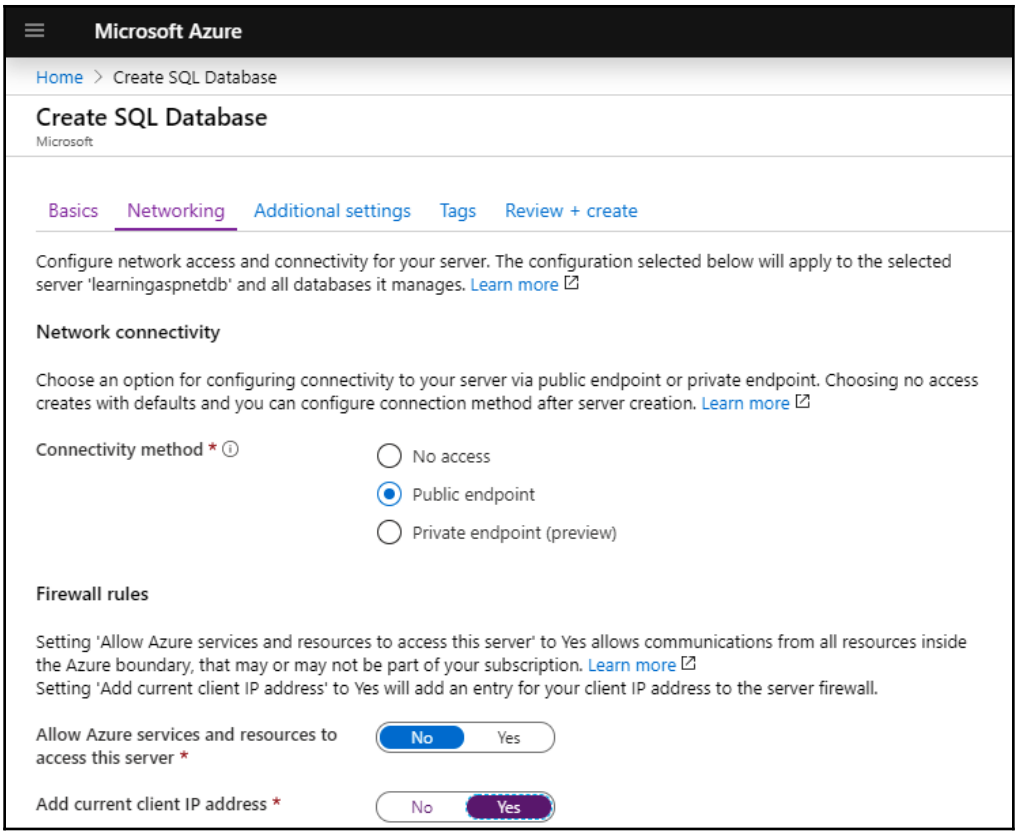
WOW! eBook
www.wowebook.org

3. Enter some values for **Server name**, **Server admin login**, and **Password**, then click on the **OK** button, as shown here:

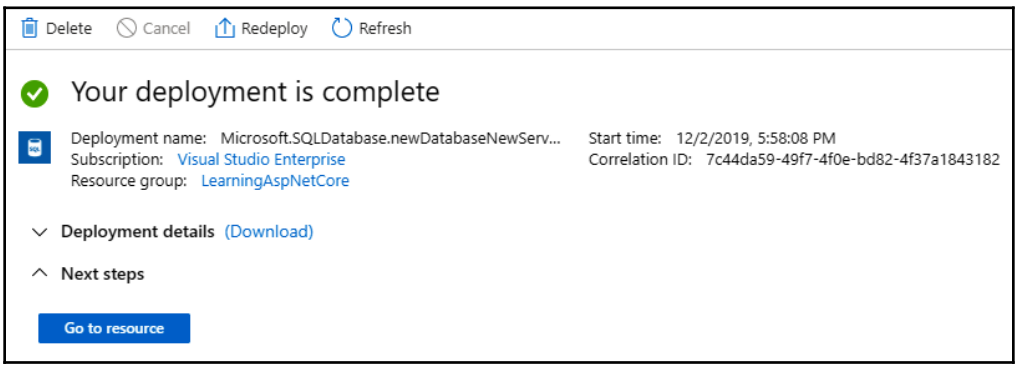


4. Click **Next: Networking** from the preceding screen, to configure in the following screen whether the public is allowed to access this database. For the purposes of our application, we need it to only be accessed internally, as follows:

WOW! eBook
www.wowebook.org

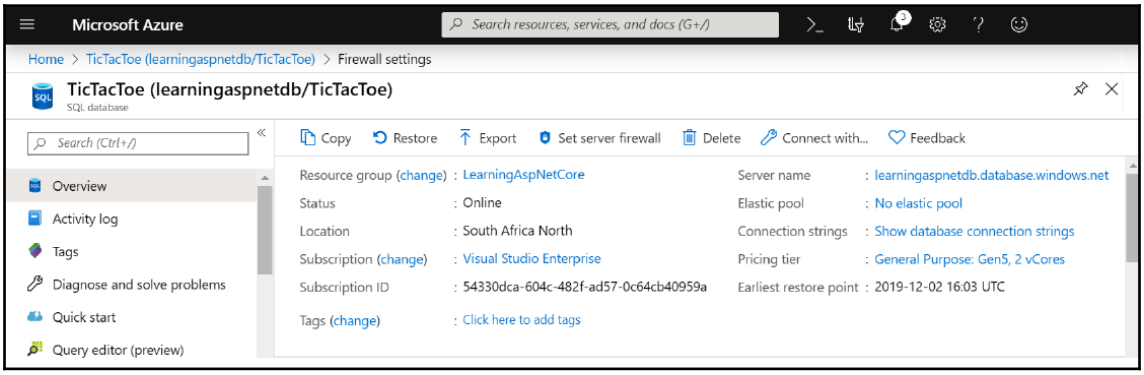


5. Click **Review + create** and eventually **Create**, then you will see a final screen similar to the following one if your deployment was successful:

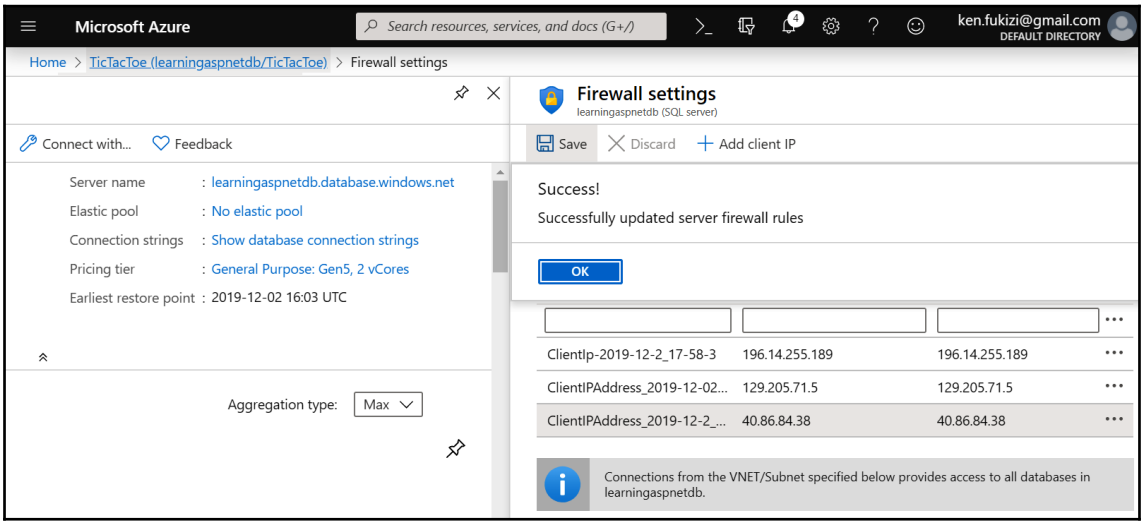


WOW! eBook
www.wowebook.org

6. You will need to allow access to the SQL database to execute the database generation scripts for the `TicTacToe` application. In the left-hand drop-down menu, or from the home page, click on **SQL databases** and select the `TicTacToe` database, as shown in the following screenshot:

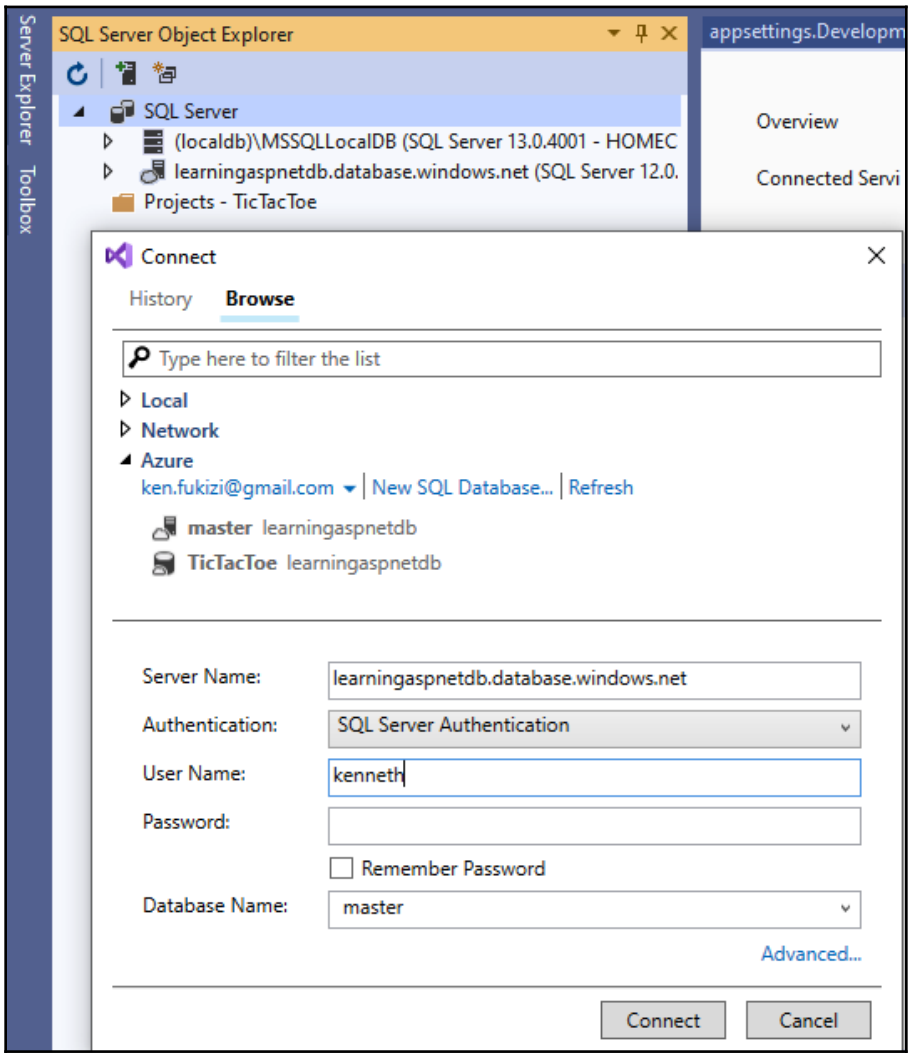


7. Click on **Set server firewall** from the preceding screen to be able to add a new rule allowing access to the SQL database from your IP. Click on **Add client IP**, verify your IP, and click on **Save** to add the new rule, which results in the following screen:



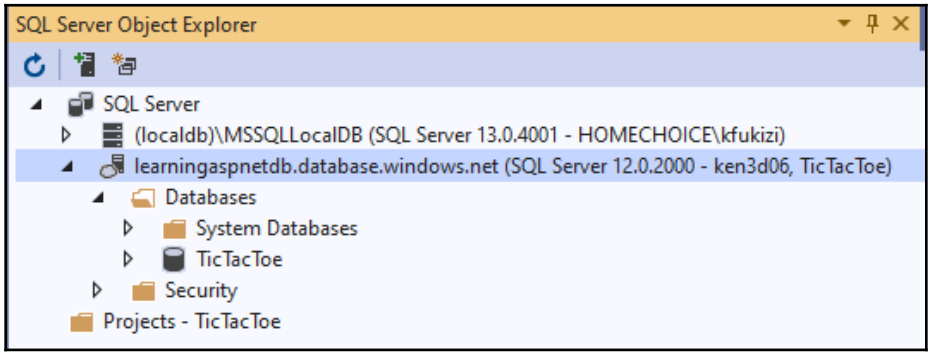
WOW! eBook
www.wowebook.org

8. Open Visual Studio 2019, go to the **SQL Server Object Explorer**, and add a new **SQL Server**, using the connection information from the `TicTacToe` Azure database connection string. You will find that this has been filled in automatically for you if you click on the Azure link, as follows:

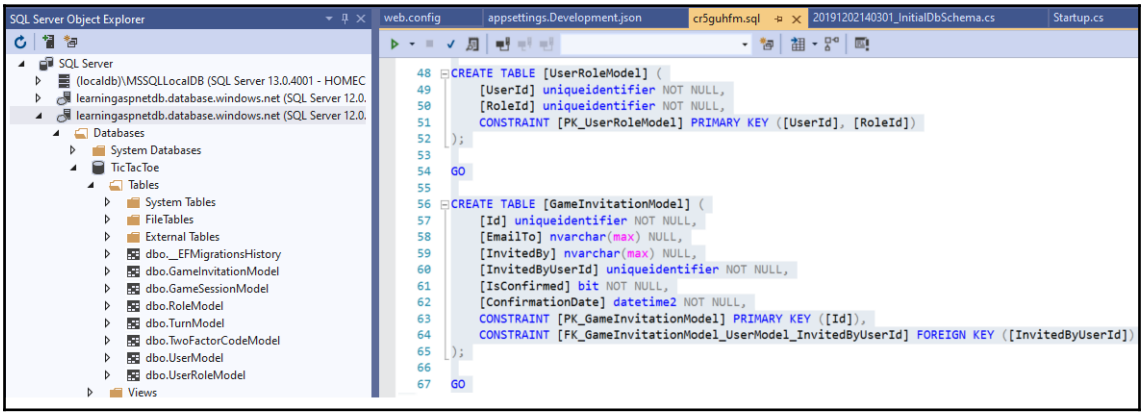


WOW! eBook
www.wowebook.org

9. Add a new database to the Azure SQL Server instance, as you would have done in the AWS example; it will be used to execute the `TicTacToe` database generation scripts, as follows:



10. If you did not follow the AWS example, open the **Package Manager Console** in Visual Studio 2019 and execute the Script-Migration instruction; otherwise, you can reuse the same scripts, as in the previous example.
11. Take the generated script and copy it into a query window for the Azure `TicTacToe` database, then execute the script to create the database and the various database objects, as follows:



12. If you did not go through the AWS deployment example, please remember to add the same `web.config` file as in the preceding AWS example, and add `webBuilder.UseIISIntegration()` to the `CreateHostBuilder` method, in `Program.cs`, since App Service is based on IIS as the host for .NET Core applications.

WOW! eBook
www.wowebook.org

At this point, you have done enough in terms of deployment. All the code files are in, and the database is set up and connected. We will look at what to do next to make sure that users are able to see a running application, in the next—and final—chapter. But meanwhile, if you found deployment using CI/CD tools with Azure DevOps a little involved, there's an easier option for you, which will be discussed in the next section: *Deployment through the Web Deploy tool*. Stay focused.

Deployment through the Web Deploy tool

The previous example used deployment through the Azure DevOps CI/CD functionality. Alternatively, you can use **Web Deploy** to publish your project to Azure, directly from Visual Studio 2019, so let's do exactly that—prepare the application and deploy it via Visual Studio 2019 into the Microsoft Azure App Service instance you created before, as follows:

1. Since App Service is based on IIS as the host for .NET Core applications, you now have to add a `web.config` file to the `TicTacToe` project. You should, however, already have done that if you have followed the AWS example from before, as follows:

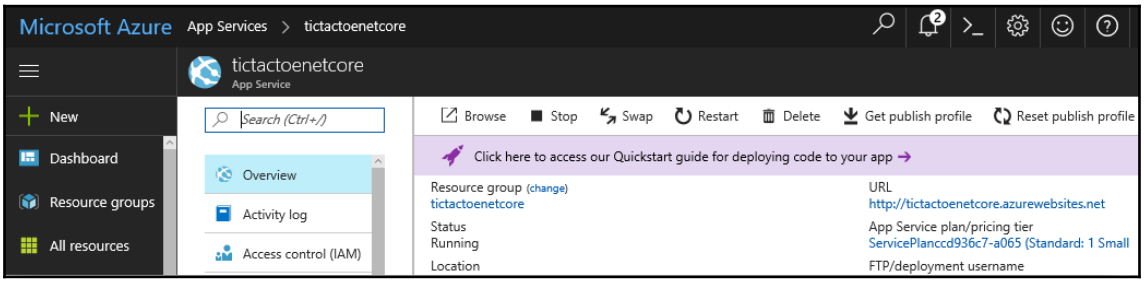
```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*"
       verb="*" modules="AspNetCoreModule"
       resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
     arguments=".\TicTacToe.dll"
     stdoutLogEnabled="true"
     stdoutLogFile=".\logs\stdout"
     forwardWindowsAuthToken="true" />
  </system.webServer>
</configuration>
```


WOW! eBook
www.wowebook.org

2. Furthermore, you have to enable IIS integration; for that, open the `Program.cs` file and change the WebHost builder configuration to enable IIS integration. You should, however, already have done that if you have followed the AWS example from before, as follows:

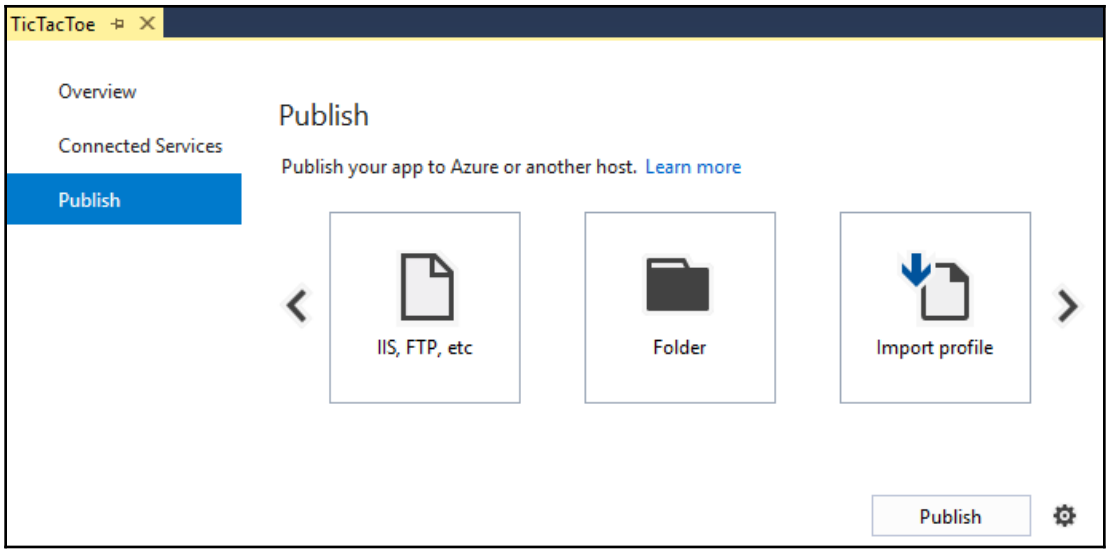
```
public static IHostBuilder CreateHostBuilder(string[] args) =>
Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
         webBuilder.UseStartup<Startup>();
         webBuilder.CaptureStartupErrors(true);
         webBuilder.PreferHostingUrls(true);
         webBuilder.UseUrls("http://localhost:5000");
         webBuilder.ConfigureLogging((hostingcontext, logging) =>
         {
             logging.AddLoggingConfiguration(hostingcontext.
              Configuration); });
              webBuilder.UseIISIntegration();
      });
```

3. Go to the Microsoft Azure management portal and click on **App Services** in the left-hand menu. Select the `TicTacToe` application you have created before, click on **Get publish profile**, and download the Azure App Service **Publish** profile, as follows:

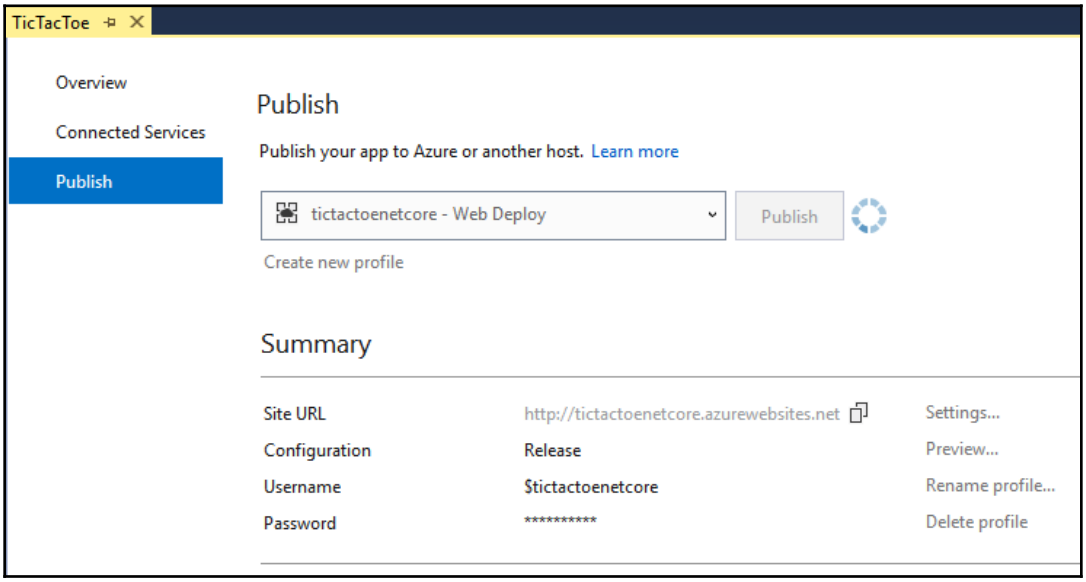


4. Right-click on the `TicTacToe` project, click on **Publish** in the context menu, then click on the **Import profile** button, as shown here:

WOW! eBook
www.wowebook.org

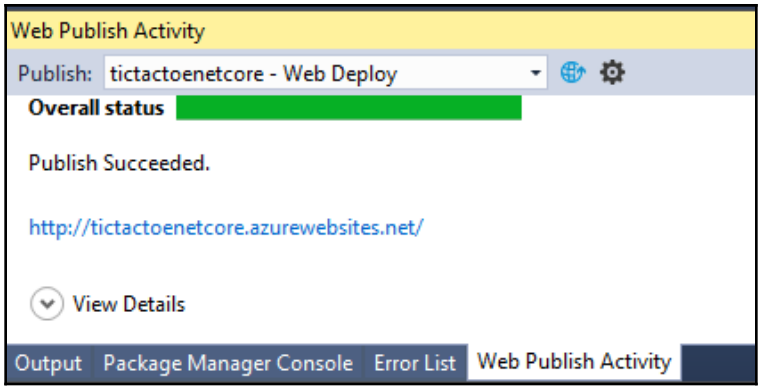


5. Select the downloaded Azure App Service **Publish** profile, and the publish process should start automatically, as follows:



WOW! eBook
www.wowebook.org

6. You can see the publish process in the **Web Publish Activity** view, as follows:



7. Open a browser and go to the application URL in Microsoft Azure, start the application, and try to register a new user.

Note that if the application is not working as expected, you will get a 404 Not Found HTTP response. Everything is working locally and the deployment in Microsoft Azure was successful, but something is wrong. You will see in the next chapter (which is about logging and monitoring) how to analyze, diagnose, understand, and fix this problem.

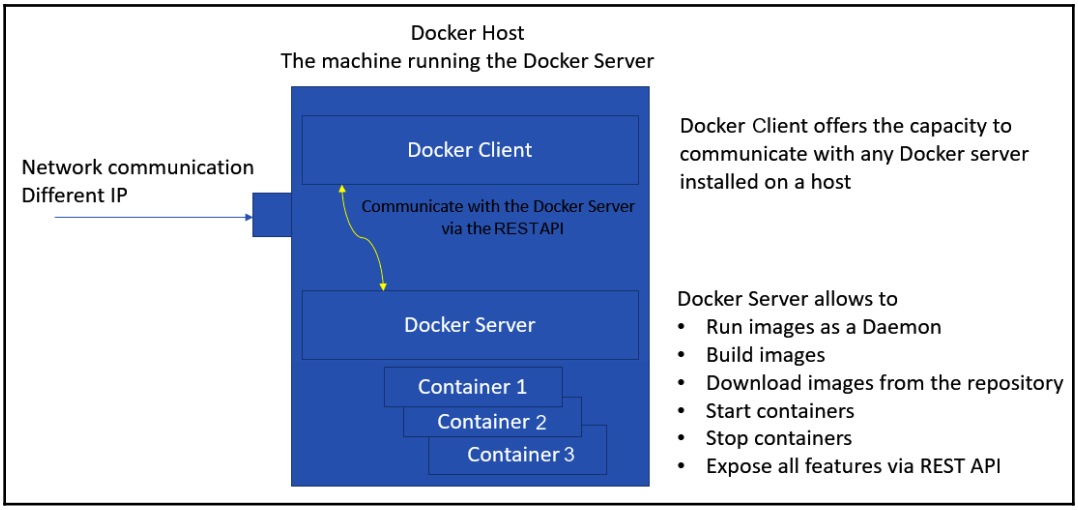
This concludes the examples for Microsoft Azure. The next section will explain how to deploy your application into Docker containers.

Deploying applications into Docker containers

Docker simplifies building, deploying, and running applications by using containers. Containers allow for the packaging of libraries, as well as any other dependencies, into a single application package (container image), which can then be shipped as a single coherent resource. This technology assures that the packaged application will run correctly anywhere the container can be used, regardless of any environment-specific settings or configurations.

WOW! eBook
www.wowebook.org

Here is a high-level schema of how Docker works:



You basically have three choices when working with Docker containers, as follows:

- Use a **Virtual Machine** (**VM**) locally or in the cloud with **Docker for Windows** or **Docker Enterprise** (Windows Server 2019 and 2016), depending on the operating system
- Use Docker Hub (`https://hub.docker.com`)
- Use either Microsoft Azure Container Service or AWS EC2 Container Service

For more information on Docker, visit the following links:

- `https://www.docker.com`
- `https://docs.microsoft.com/en-us/dotnet/core/docker/build-container`

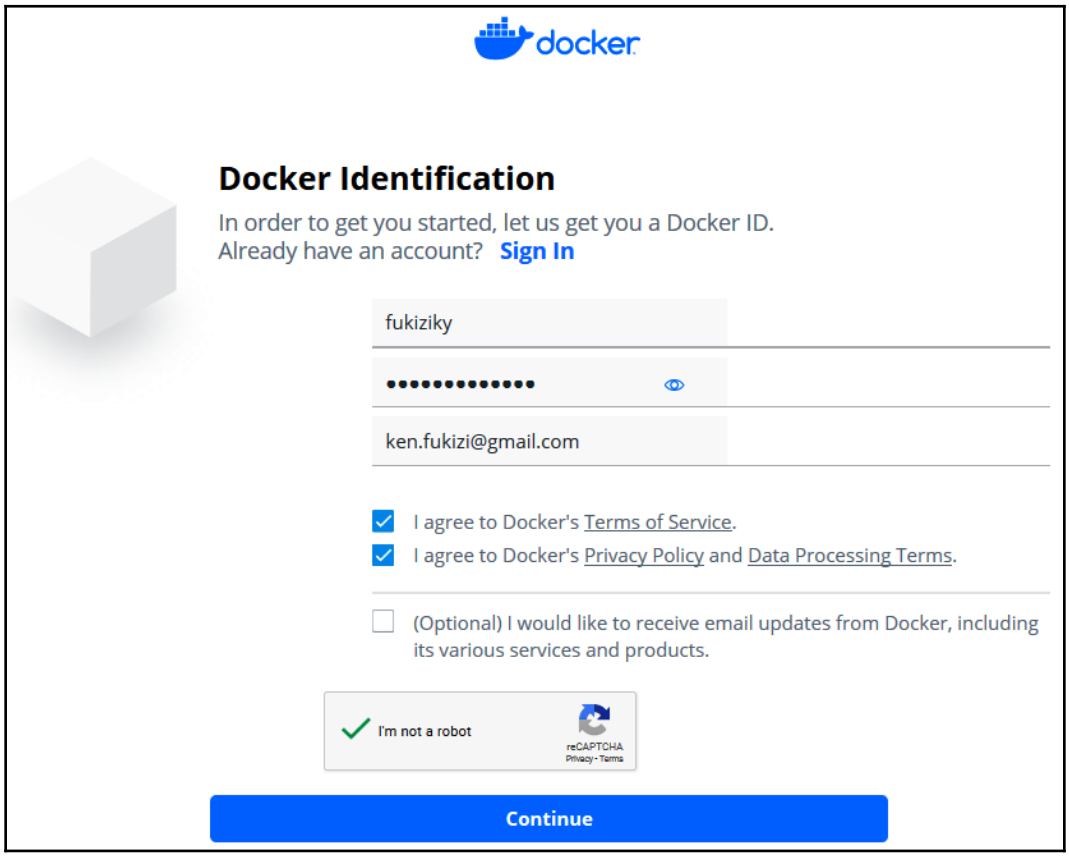
Deploying applications into Docker containers

Docker for Windows provides everything necessary to start using Docker containers in a Windows environment, whereas Docker Enterprise (Windows Server 2019 and 2016) is designed for companies that need to provide the necessary support to production environments based on the Docker technologies.

WOW! eBook
www.wowebook.org

Let's see how to use Docker in Windows and how to deploy your application in this case, as follows:

1. If you do not yet have Docker for Windows installed, go to `https://hub.docker.com`. Sign in with your Docker ID if you have one; otherwise, click on **Sign Up** and fill in your Docker ID of choice, along with a password and an email address, as follows:

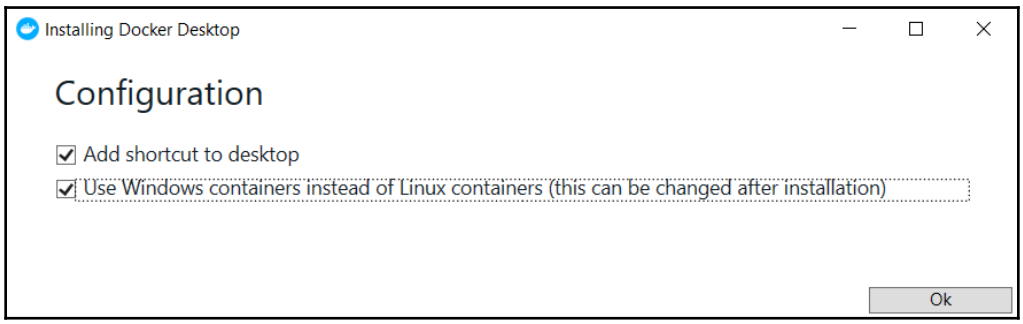


2. You will then fill in personalized details on your account, and after verification, on the welcome page, click on **Get Started With Docker Desktop**, and you will be presented with the following popup. Click on **Download Docker Desktop for Windows** and install it after the download, as follows:

WOW! eBook
www.wowebook.org

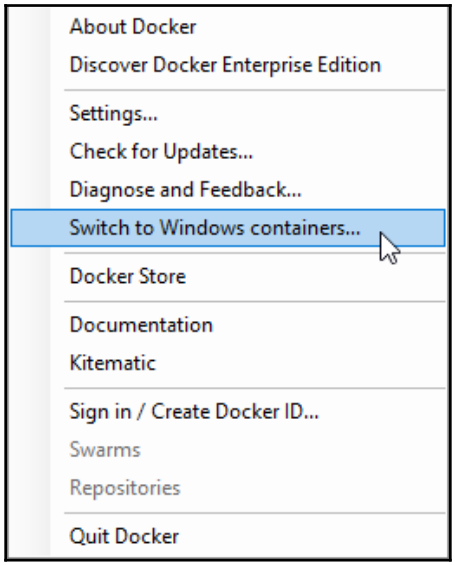
To install Docker Enterprise Edition for Windows 2016, go to `https://hub.docker.com/editions/enterprise/docker-ee-server-windows` and follow the installation instructions. After the installation, you should skip the following step, and continue directly with *Step 4*.

3. During installation, make sure you select Windows containers, as follows:

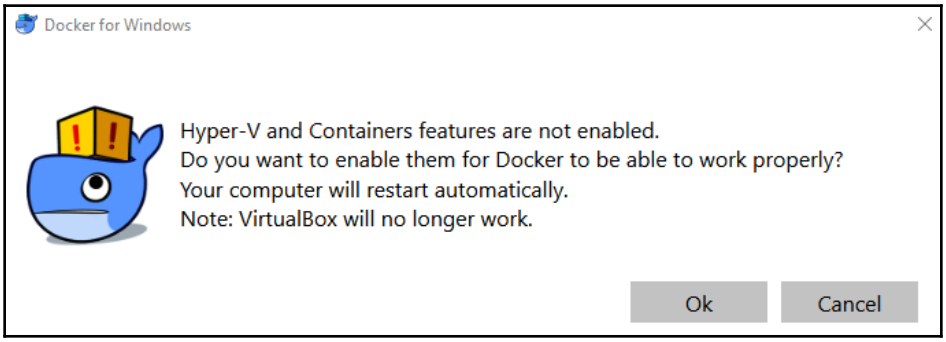


WOW! eBook
www.wowebook.org

4. You can also switch to Windows containers after installation by right-clicking on the Docker tray icon and further clicking on **Switch to Windows containers...** in the context menu, as follows:



5. If the container features have not yet been enabled in your Windows installation, Docker will ask if you would like this to be done for you. Click on the **Ok** button, as follows:



6. Open a new elevated Command Prompt with administrator rights, download the official Docker Microsoft SQL Server image, and execute the `docker pull microsoft/mssql-server-windows-express` instruction, as follows:

WOW! eBook
www.wowebook.org

```
Administrator: Command Prompt
C:\>docker pull microsoft/mssql-server-windows-express
Using default tag: latest
latest: Pulling from microsoft/mssql-server-windows-express
3889bb8d808b: Pull complete
449343c9d7e2: Pull complete
a336350523ea: Pull complete
2c12ef99f411: Pull complete
8ebb28bb3276: Pull complete
3bb8431b46ce: Pull complete
a4a938096b3c: Pull complete
656eb837f7e6: Pull complete
a3d536b0e79d: Pull complete
a0aec5f46615: Pull complete
64a0f6512541: Pull complete
b900d04a68bd: Pull complete
Digest: sha256:9fc5632b15a279097043e57b7689de29e2c441a8d7633b8c7b4e6adc9bdcfb33
Status: Downloaded newer image for microsoft/mssql-server-windows-express:latest
docker.io/microsoft/mssql-server-windows-express:latest
```

7. Download the official Docker Microsoft ASP.NET Core image, and execute the `docker pull microsoft/aspnetcore` instruction, like this:

```
Administrator: Command Prompt
C:\>docker pull microsoft/aspnetcore
Using default tag: latest
latest: Pulling from microsoft/aspnetcore
e46172273a4e: Pull complete
1f7d468f830c: Pull complete
64bca589bec8: Pull complete
d4da7ec5d06f: Pull complete
d051ed95a99c: Pull complete
5ad7c5736855: Pull complete
727d138267d5: Pull complete
27470bc34f6e: Pull complete
01d43752affd: Pull complete
74d784d1700a: Pull complete
Digest: sha256:8bd7808981568a92b7f1d0aa4bab07ef811e6417898847d9caf698a4a788ab11
Status: Downloaded newer image for microsoft/aspnetcore:latest
docker.io/microsoft/aspnetcore:latest
```


WOW! eBook
www.wowebook.org

8. To be able to compile and publish applications from Visual Studio 2019 directly into Docker, you will also need to download the specific build image and execute the `docker pull microsoft/aspnetcore-build` instruction, as follows:

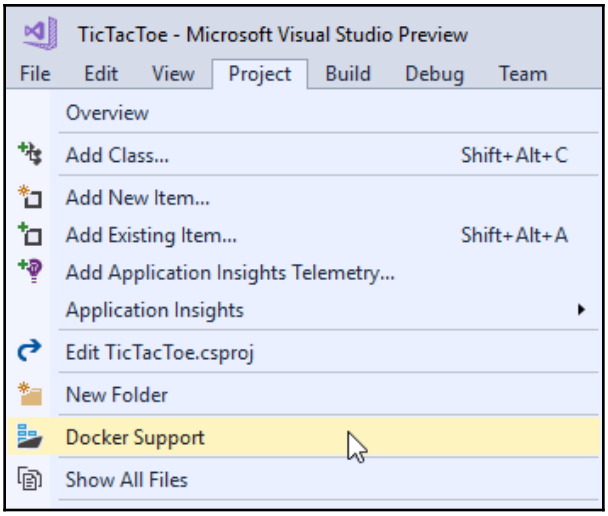
```
Administrator: Command Prompt

C:\>docker pull microsoft/aspnetcore-build
Using default tag: latest
latest: Pulling from microsoft/aspnetcore-build
e46172273a4e: Already exists
1f7d468f830c: Already exists
f5ee64baf8ad: Pull complete
2119da921f3a: Pull complete
4192e27358f1: Pull complete
27bdc0c16145: Pull complete
025bd5d80cc3: Pull complete
18acd06947c5: Pull complete
37090235eb0f: Pull complete
8509ed346dbe: Pull complete
05c389c9cad3: Pull complete
4ae25802fca9: Pull complete
c183c7bc0979: Pull complete
11bcd918c41e: Pull complete
a70bdf1efa27: Pull complete
15cd81064a9a: Pull complete
00b0700a7d48: Pull complete
71c0c8e9b199: Pull complete
Digest: sha256:ab861527a8485e7df91069e80cd7a94237c22995f13494c2cccc071b76e347f0
Status: Downloaded newer image for microsoft/aspnetcore-build:latest
docker.io/microsoft/aspnetcore-build:latest

C:\>
```

9. If you check on the Docker images installed so far, you should have the following:

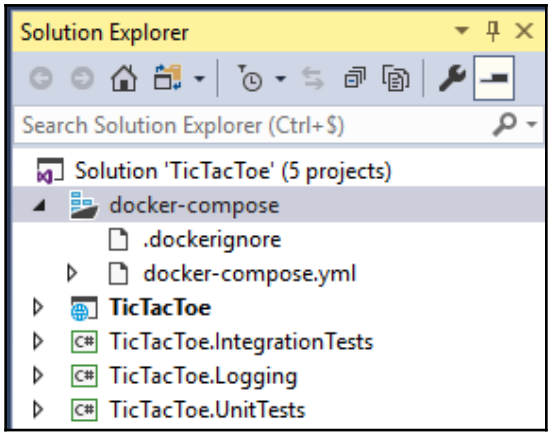
```
Administrator: Command Prompt

C:\>docker images
REPOSITORY                          TAG       IMAGE ID       CREATED          SIZE
microsoft/aspnetcore-build          latest    45d1a4544519   14 months ago    2.07GB
microsoft/aspnetcore                latest    3b51a44dad60   14 months ago    536MB
microsoft/mssql-server-windows-express  latest  1986b8a8f950   21 months ago    13.8GB

C:\>
```


WOW! eBook
www.wowebook.org

10. Open Visual Studio 2019, then open the `TicTacToe` project; in the menu, click on **Project | Docker Support** and select the **Windows** operating system, as follows:



11. A new project called `docker-compose` will be autogenerated and added to the solution; it should contain a `.dockerignore` file (files to be ignored during deployment) and a `docker-compose.yml` file (deployment instructions), as follows:



WOW! eBook
www.wowebook.org

12. Update the `docker-compose.yml` file in the **Docker Compose Project**, like this:

```
version: '3'
services:
  sql:
    image: "microsoft/mssql-server-windows-express"
    environment:
      sa_password: "123TicTacToe!"
      ACCEPT_EULA: "Y"
  tictactoe:
    image: tictactoe
    build:
      context: .
      dockerfile: TicTacToe\Dockerfile
    ports:
      - "8081:5000"
    depends_on:
      - sql
```

13. Update the `DefaultConnection` in the `appsettings.json` file in the `TicTacToe` application, as follows:

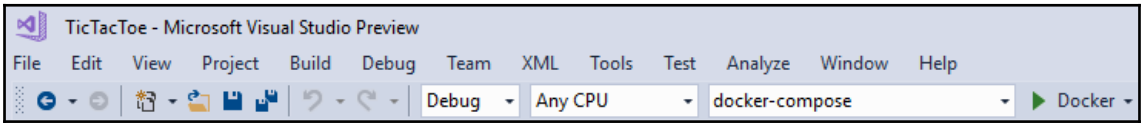
```
"DefaultConnection":
 "Server=sql;Database=Master;MultipleActiveResultSets=true;
  User id=sa;pwd=123TicTacToe!"
```

14. Update the `Program.cs` file in the `TicTacToe` project; remove the IIS Integration, because the Docker ASP.NET Core image is based on Kestrel instead of IIS, as follows:

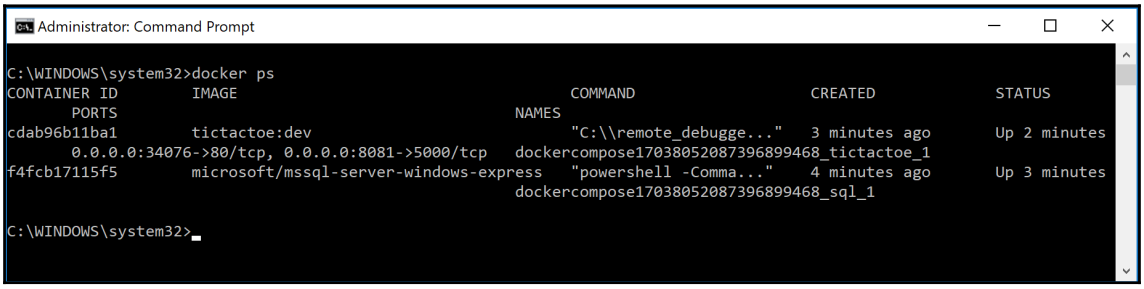
```
public static IHostBuilder CreateHostBuilder(string[] args) =>
Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
         webBuilder.UseStartup<Startup>();
         webBuilder.CaptureStartupErrors(true);
      });
```


WOW! eBook
www.wowebook.org

15. Start the application by pressing *F5* (the `docker-compose` project should be set as a startup). The application should now have been automatically deployed into a Docker container; verify that everything is still working as expected, as follows:



16. Open Command Prompt and execute the `docker ps` instruction, to see all running Docker processes. There should be multiple running container instances, as follows:



In this case, we have an instance of the **tictactoe** application image for the **dev** environment, and the **microsoft/mssql-server-windows-express** version. In the next section, we will see how to publish such images to Docker Hub.

Publishing images to Docker Hub

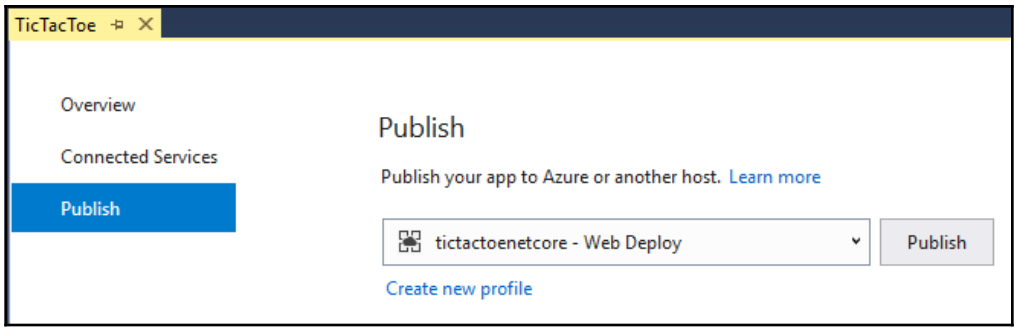
You can upload your application images to the central cloud-based Docker repository called Docker Hub, and then use them in Microsoft Azure, AWS, or any other Docker-supported environments.

Note that there are also other Docker registries you could use, such as **Azure Container Registry** and others. Since Docker provides its own registry via Docker Hub, it is, however, advised to use that.
For more information on Docker Hub, check out `https://docs.docker.com/docker-hub`.

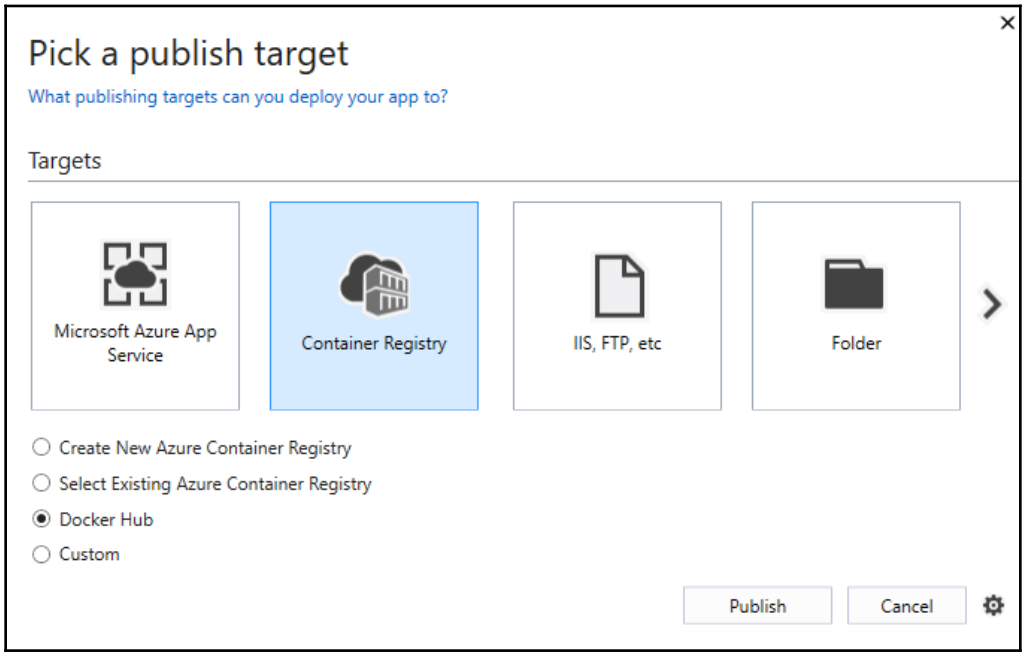
WOW! eBook
www.wowebook.org

The following example showcases how to publish and upload the sample `TicTacToe` application to Docker Hub:

1. Right-click on the `TicTacToe` project and select **Publish** in the context menu; since you have already created a publish profile in the preceding examples, you have to add a new one. Click on **Create new profile**, as follows:

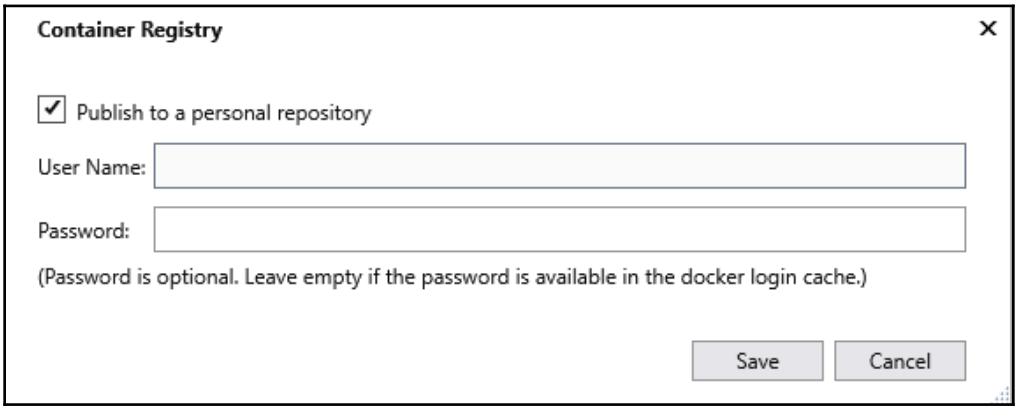


2. Click on the **Container Registry** button, select **Docker Hub**, and click on the **Publish** button, as shown in the following screenshot:

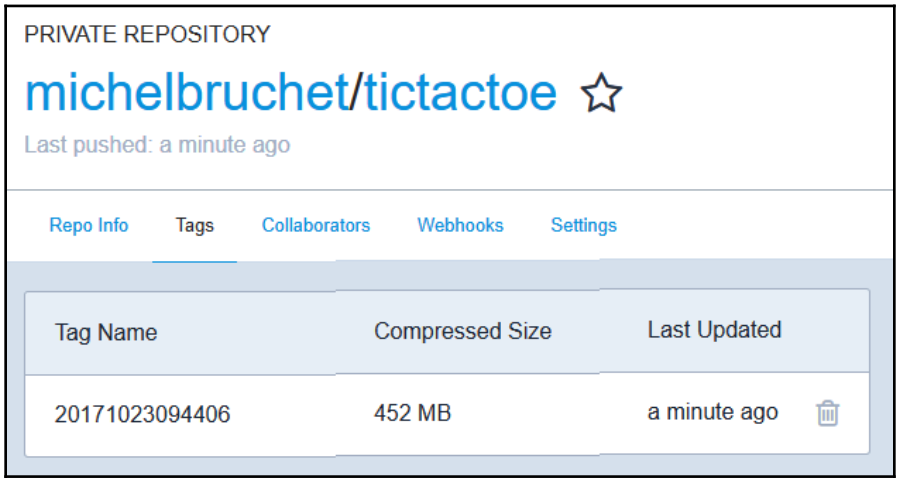


WOW! eBook
www.wowebook.org

3. Enter your Docker Hub **User Name** and **Password**, and click on **Save**, as shown in the following screenshot:



4. Your container image will be published to Docker Hub; when it has been finished, go to Docker Hub and verify that the image has been uploaded, as follows:



Well done for getting this far, and on the application development side of things, we have pretty much looked at just about every angle of how to come up with a world-class application. You should pat yourself on the back for finally having your application in a container hub.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we talked about the various options you have when it comes to hosting and deploying your ASP.NET Core 3 web applications. You have learned what hosting is, and how to choose the appropriate solutions for a given use case. This will allow you to make better decisions for your own applications.

You have seen how to sign up for an AWS account, how to provision the technical environment, and how to deploy ASP.NET Core 3 web applications. Furthermore, you have seen how to sign up for a Microsoft Azure account, how to provision the technical environment, and how to deploy ASP.NET Core 3 web applications using this powerful public cloud computing platform.

We then talked about Docker and the various deployment choices you have when you use this modern, increasingly adopted, and impactful technology. You are well prepared for the future, since Docker may well completely change our way of thinking concerning deploying and managing applications.

You have gained very important skills any serious developer should have, which include hosting your web applications, deploying to Docker containers, and deploying to AWS or Microsoft Azure; and, last but not least, you have gained skills required in continuous deployment.

In the next chapter, we will explain how to manage and supervise deployed web applications efficiently, which is very important for a DevOps approach.

WOW! eBook
www.wowebook.org

13
Managing ASP.NET Core 3 Applications

After having finished the development life cycle, we could have stopped there. However, this last chapter has been added, to underline the importance of a thorough **DevOps** approach.

For now, we have only talked about the **development** (**Dev**) side, but you should also embrace the **operations** (**Ops**) side in DevOps, which consists of managing and supervising your applications during runtime.

This very important subject is often underestimated and, even worse, is sometimes completely left aside. Developers tend to think that it is not a part of their job. They often say things such as: *But it works on my machine*, and: *This is your problem, not mine*. This is also commonly called the **wall of confusion**. Agile methodologies and DevOps aim to avoid this kind of thinking, and this chapter will give you some advice and examples on how to better address those issues within your ASP.NET Core 3 applications.

The success of your application will depend on how you can help IT operations understand what is happening during runtime. This means providing them with the means to manage and supervise applications quickly and efficiently.

Only then will you be able to provide high-quality applications with a low **mean time to repair** (**MTTR**) for bugs, which can make a difference in becoming a future market leader within your specific market.

Furthermore, it is easy for you to address these subjects when using ASP.NET Core 3, since most of the time, you can take advantage of integrated or provided features, without the need for any bigger code changes.

We will start by having a look at how we can add logging for both Azure and **Amazon Web Services** (**AWS**), and then we will take a look at how we can monitor the application on-premises and in Docker, before looking at how we can monitor in Azure and AWS.

WOW! eBook
www.wowebook.org

In this chapter, we will cover the following topics:

- Logging in ASP.NET Core 3 applications
- Monitoring ASP.NET Core 3 applications

Logging in ASP.NET Core 3 applications

In `Chapter 12`, *Hosting ASP.NET Core 3 Applications*, we explained how to deploy your ASP.NET Core 3 applications to Microsoft Azure, AWS, and Docker. Let's go further, and understand how to add logging and monitoring to these environments, which are important for diagnosing unexpected behavior and errors.

First, some theoretical background, and then some practical examples. Are you ready to learn what it takes to help IT operations? Come on; it's the last chapter. Let's go!

Logging within applications consists of creating data to help understand what is happening during runtime. Several types of messages can be logged, such as information, warnings, and errors.

This data should then be persisted to log files, databases, SaaS solutions, or other destinations. To improve application performance, it is recommended to allow IT operations to change the level of verbosity of the collected logging data during application runtime. For instance, in production environments, only warnings and errors should be logged, while it makes perfect sense to enable more efficient logging of everything during development time and to better understand what exactly is happening behind the scenes.

It is advisable to use a standard framework such as **Event Tracing for Windows (ETW)** to structure and format logging data so that IT operations can use their preferred monitoring tools to quickly and easily read and diagnose reasons for errors. Famous logging frameworks such as **Serilog** or **Log4net** also support standard output formats, so you could also use them if you like.

So, let's look at some concrete examples on how to handle logging for your ASP.NET Core 3 applications, in different environments such as on-premises, in the public cloud, and in Docker.

In on-premises environments, logging data is stored in a log file most of the time. In this case, the application needs to have write access to write to the log file, and it is recommended to store all log files in a central folder called `logs` under the application path.

WOW! eBook
www.wowebook.org

In Microsoft Azure, you basically have three different solutions to handle logging within your applications, as follows:

- **Standard file logging**: This is the easiest method, without any code modifications, but it is also the least powerful. You need to download files to retrieve logging data for your application.
- **Azure App Service Diagnostics**: This is the recommended solution if you have no more than a single instance for your application service since there are no log centralization features provided.
- **Azure Application Insights**: This is the most integrated and most powerful solution, which works across all application layers.

AWS provides CloudWatch for logging and monitoring. The logging mechanisms provided are very similar to those for Microsoft Azure. When you have understood how these work in Microsoft Azure, you will be able to apply your knowledge to AWS easily and quickly, as you will see in the examples provided.

For more information, you can visit the AWS CloudWatch website at `https://aws.amazon.com/en/cloudwatch`.

Docker does not provide any of the integrated monitoring or logging services that exist for Microsoft Azure or AWS. This means that for adding, logging, and monitoring functionalities to your ASP.NET Core 3 applications in Docker, you have to use a log file. Furthermore, you have to provide your own centralized log recovery and analysis mechanisms to get consistent logging and monitoring data.

However, since applications can be instantiated multiple times, this may not be the best approach. Instead, you could also directly log to a centralized console, which should be the most efficient and most appropriate solution in a Docker environment.

Logging in Microsoft Azure

OK; now that you have seen several solutions for logging in different environments, we will focus on Microsoft Azure. What happens if you take on the role of IT operations, which need to diagnose why an application is not working as expected in Microsoft Azure? What are your choices, and what would be the best solution? That is exactly what you will learn in this section.

WOW! eBook
www.wowebook.org

If you remember, we have already talked about logging on an application level in `Chapter 4`, *Basic Concepts of ASP.NET Core 3 via a Custom Application: Part 1*, of this book. There, we added logging application events to a log file in a `logs` subfolder of the application folder. This folder needs to be synchronized and monitored for disk space usage because, when it gets too big, it may well become a reason for failure in itself.

Furthermore, there are multiple sources of logs, since application logs and environmental logs (**Internet Information Services** (**IIS**), Windows, SQL Server, and so on) are handled separately. You have to combine all the information to get a holistic view of what is happening behind the scenes. This is very complicated and very time-consuming.

As you can see, it requires a lot of manual work to read and analyze application logs in this case. This becomes even more of an issue if you need to monitor and supervise a high number of applications at the same time. Doing everything manually is not really an option. We need to find a better solution.

Moreover, there are better and more integrated solutions in Microsoft Azure! If you deploy your applications in Azure App Service, for instance, you can use the Azure App Service Diagnostics. This feature can be enabled directly from the portal. Additionally, application logs and environmental logs are automatically centralized in a single place, which helps to find problems in a much quicker and more straightforward way.

Enabling Microsoft Azure App Service

Enabling Microsoft Azure App Service Diagnostics is very easy, so let's see how to do that now:

1. Open the Tic-Tac-Toe web project in Visual Studio 2019, and add a new extension called `AzureAppServiceDiagnosticExtension` to the `Extensions` folder, as follows:

```
public class AzureAppServiceDiagnosticExtension
{
  public static void AddAzureWebAppDiagnostics
 (IConfiguration configuration, ILoggingBuilder
    loggingBuilder)
  {
    loggingBuilder.AddAzureWebAppDiagnostics();
  }
}
```


WOW! eBook
www.wowebook.org

2. Update the `AddLoggingConfiguration` method in the `ConfigureLoggingExtension` class, and add a case for the newly added **Azure** `ApplicationServiceDiagnosticExtension` from before, as follows:

```
foreach (var provider in loggingOptions.Providers)
{
  switch (provider.Name.ToLower())
  {
    case "console":  {  loggingBuilder.AddConsole();  break; }
    case "file":   {  ...  }
    case "azureappservices":
    {
      AzureAppServiceDiagnosticExtension
.AddAzureWebAppDiagnostics(configuration,loggingBuilder);
      break;
    }
    default:  {  break;  }
  }
}
```

3. Update the `appsettings.json` configuration file, and add a new provider for Azure App Service, as follows:

```
"Logging": {
  "Providers": [
    {
      "Name": "Console",
      "LogLevel": "1"
    },
    {
      "Name": "File",
      "LogLevel": "2"
    },
    {
      "Name": "azureappservices"
    }
  ],
  "MinimumLevel": 1
}
```

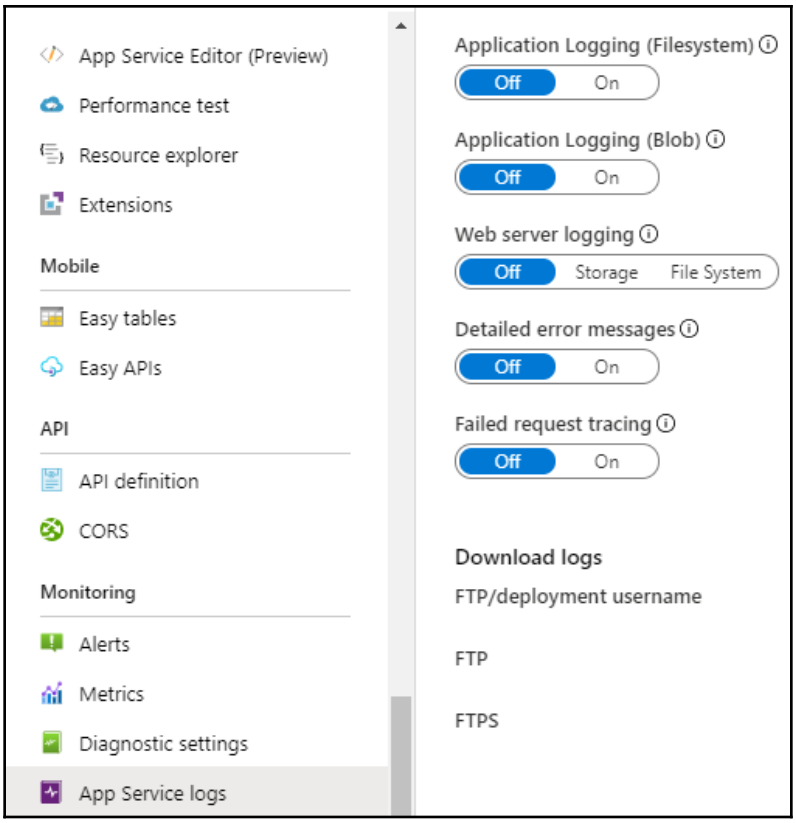
4. Update the `Program.cs` file, change the WebHost builder configuration to enable IIS integration, and add the logging configuration, as follows:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
```


WOW! eBook
www.wowebook.org

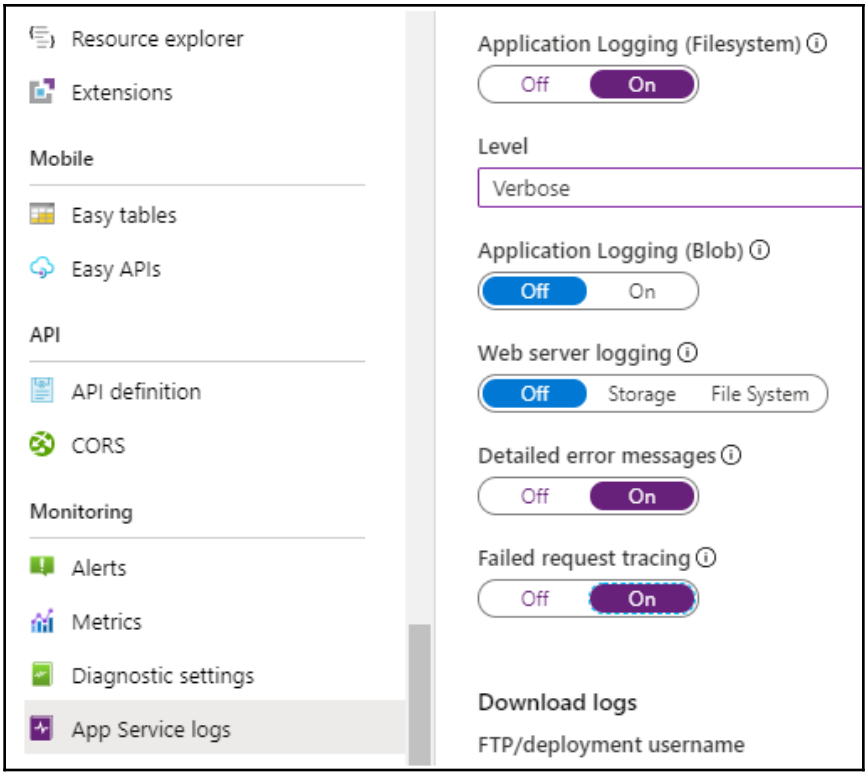
```
{
    webBuilder.UseStartup<Startup>();
    webBuilder.CaptureStartupErrors(true);
    webBuilder.PreferHostingUrls(true);
    webBuilder.ConfigureLogging((hostingcontext, logging) =>
    {
        logging.AddLoggingConfiguration(hostingcontext.
        Configuration); });
    webBuilder.UseIISIntegration();
});
```

5. Publish the Tic-Tac-Toe web application to Azure App Service. If you do not know how to do that, you can look it up in `Chapter 12`, *Hosting ASP.NET Core 3 Applications*.
6. Go to the Microsoft Azure portal website, click on **App Services** in the menu, select the **Tic-Tac-Toe App Service** you have deployed, and scroll down until you see the **Monitoring** section, shown in the following screenshot:



WOW! eBook
www.wowebook.org

7. In the **Monitoring** section, click on **App Service logs**, and then set the **Application Logging (Filesystem) On** button. Select **Level** as **Verbose**, enable **Detailed error messages** and **Failed request tracing**, and then click on the **Save** button, as follows:



The Tic-Tac-Toe application will now start logging data in the Azure App Service filesystem. However, this is only the first step. You will need to retrieve the logs to be able to analyze them.

There are multiple ways of accessing the logs, depending on your specific needs. Some of them are specified here, as follows:

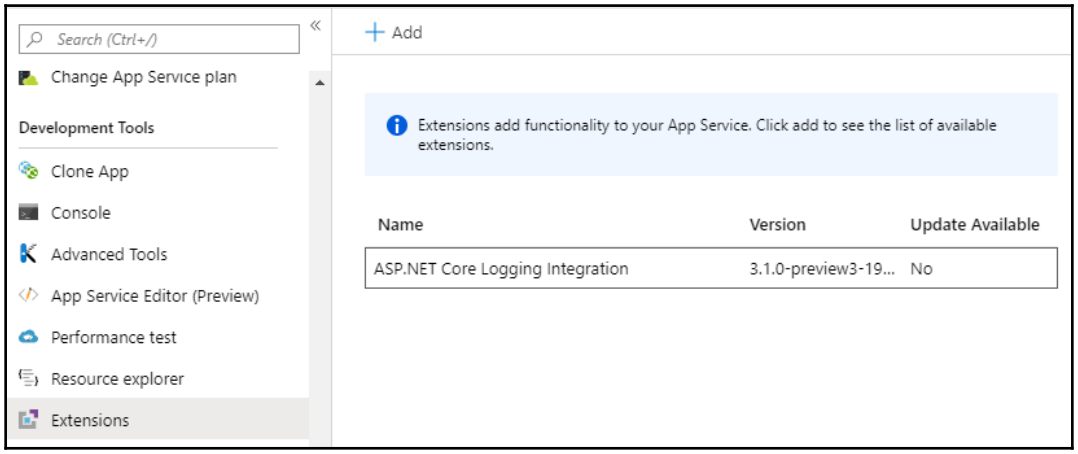
- Using FTP or FTPS to browse the `logs` folder
- Configuring Azure Blob Storage and then downloading the blob content, which also has the benefit of centralizing logs for multiple services in a single place
- Using a dedicated application to retrieve logs automatically

WOW! eBook
www.wowebook.org

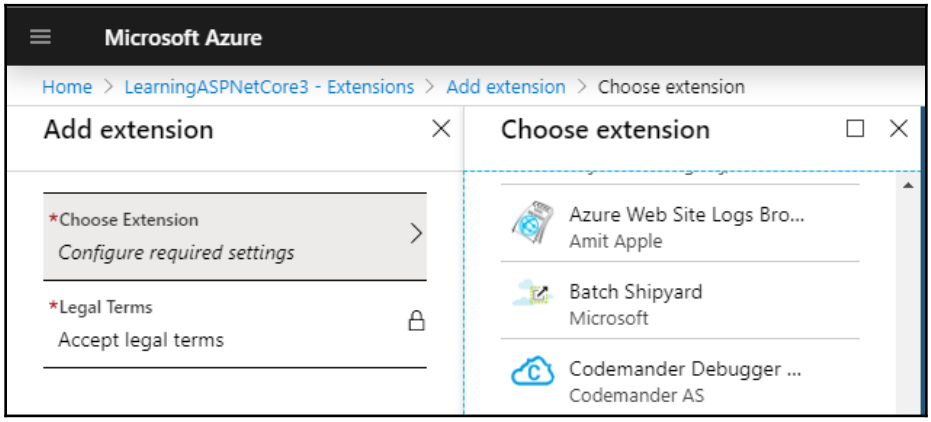
Fortunately, the community has already worked on an open source solution on GitHub, called the **Azure Web Site Logs Browser** extension, which you can use. This solution consists of adding an extension to your Azure portal.

You will now see how to add the **Azure Web Site Logs Browser** extension to the Microsoft Azure portal to analyze logs by following these steps:

1. Go to the Microsoft Azure portal website, click on **App Services** in the menu, select the **Tic-Tac-Toe App Service** you have deployed in the preceding example, scroll down until you see the **Development Tools** section, click on **Extensions**, and then on the **Add** button, as shown in the following screenshot:

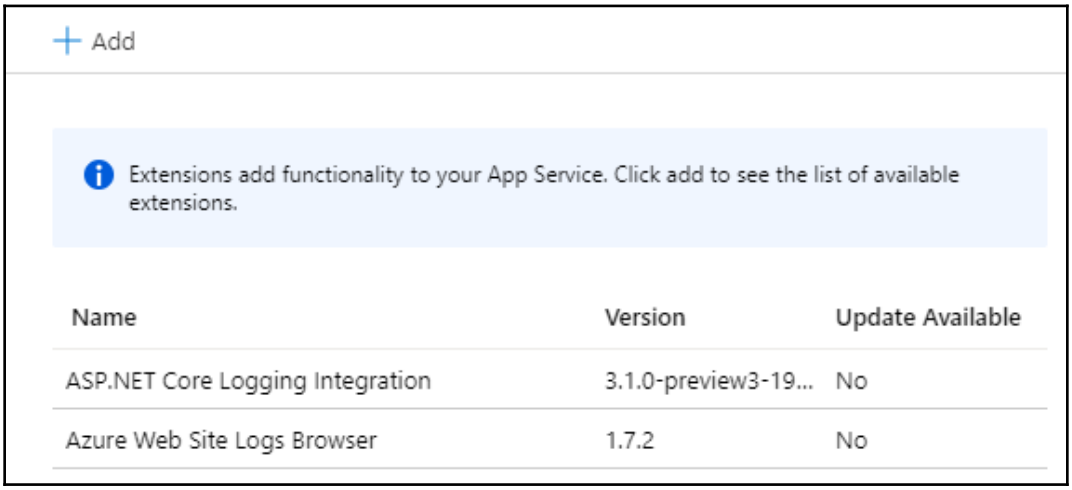


2. Select and install the **Azure Web Site Logs Browser** extension, published by **Amit Apple**, as follows:

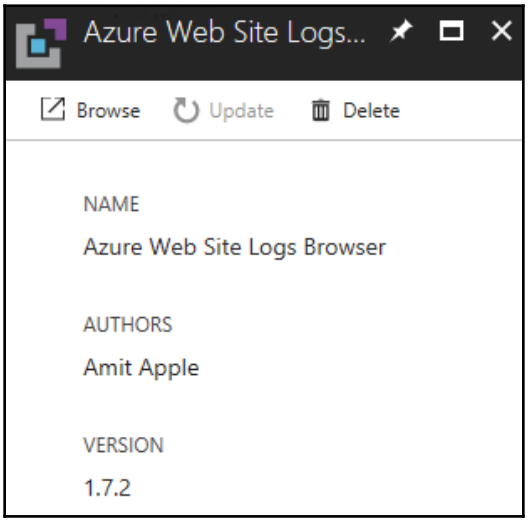


WOW! eBook
www.wowebook.org

3. Once installation has finished, the extension will be added to the active extensions for your **Tic-Tac-Toe App Service**, as follows:

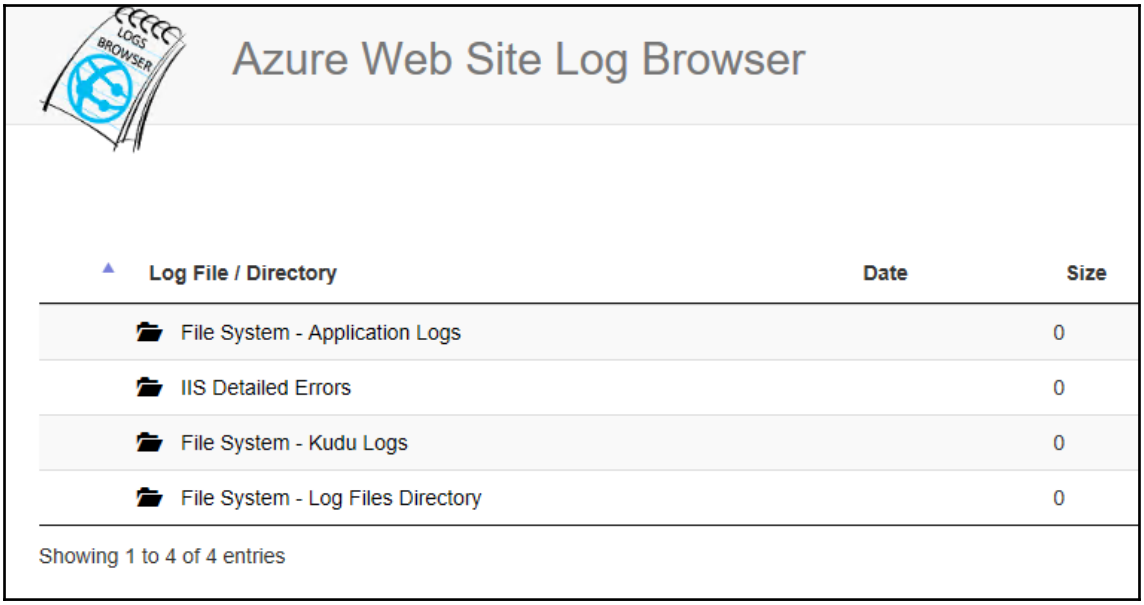


4. Click on the **Azure Web Site Logs Browser** extension, and you will see an overview with the extension name, its author, and version number, as well as other additional information. Click on the **Browse** button, as shown in the following screenshot:

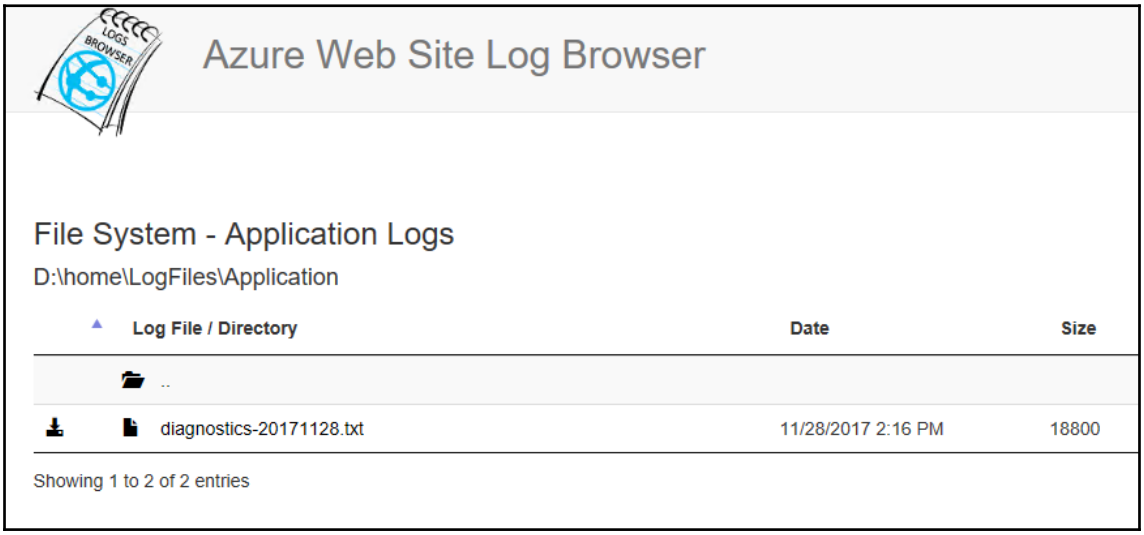


WOW! eBook
www.wowebook.org

5. A new browser window will be opened automatically, where you can see different log file sources. Click on **File System - Application Logs**, as shown in the following screenshot:

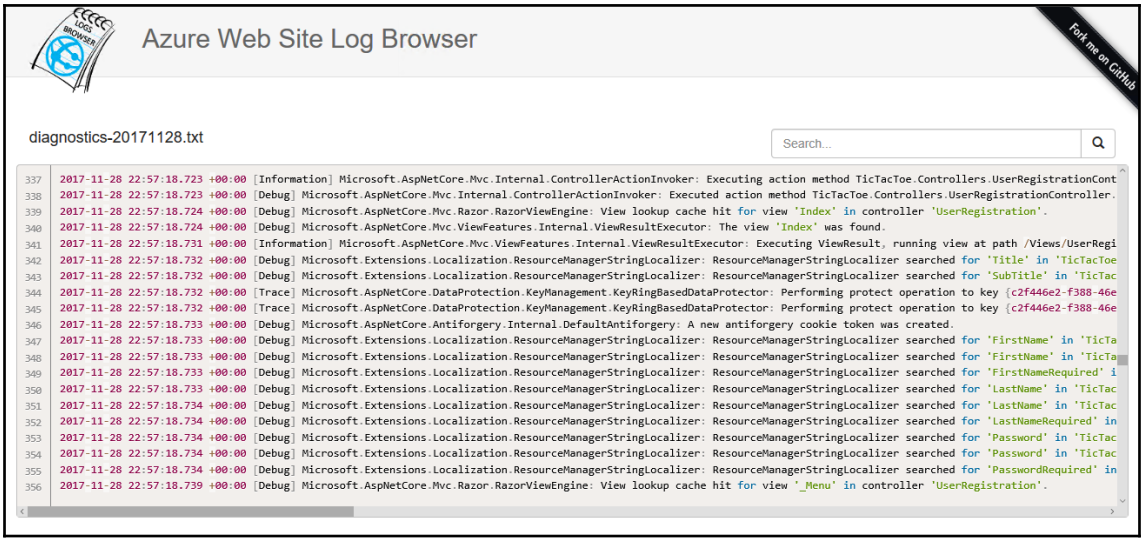


6. Select a log file with the diagnostic data you need to analyze, as follows:



WOW! eBook
www.wowebook.org

7. Read and scroll through the color-coded log file content. You will automatically see generated log entries, as well as log entries you have added by yourself in the preceding chapters, as follows:



That is all you need to know in order to have meaningful logs and be able to view them. Logs are quite important for every application and, if designed properly, they can save you a lot of time and effort and, in turn, money, which you could potentially lose if, for example, it took you a long time to find an anomaly because of insufficient logging. Let's now look at how we can do the same in AWS in the next section.

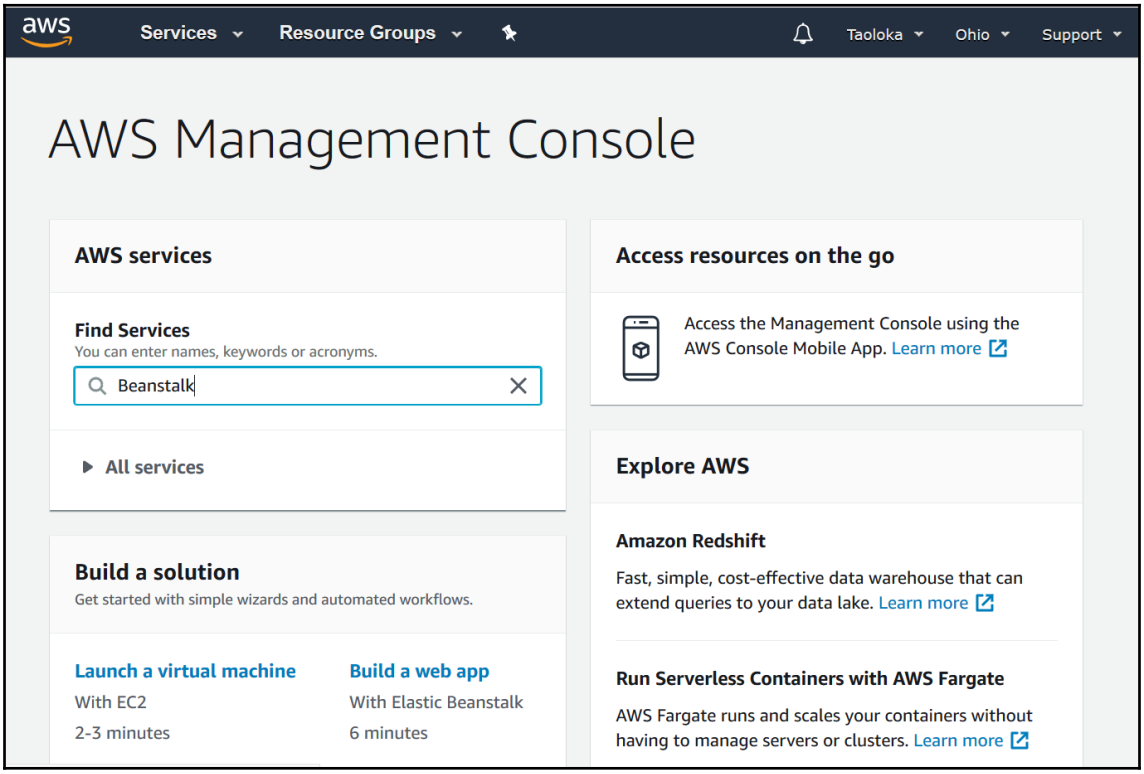
Logging in AWS

If you are using AWS, then adding logging to your ASP.NET Core 3 application will be very straightforward for you. You just have to write your application logs to the console, and the applications—which are deployed in AWS Elastic Beanstalk—will automatically store their logs in AWS CloudWatch. You will then be able to use the CloudWatch dashboard to analyze what is happening. This is comparable to Application Insights and its dashboard, which you have seen in the preceding example.

WOW! eBook
www.wowebook.org

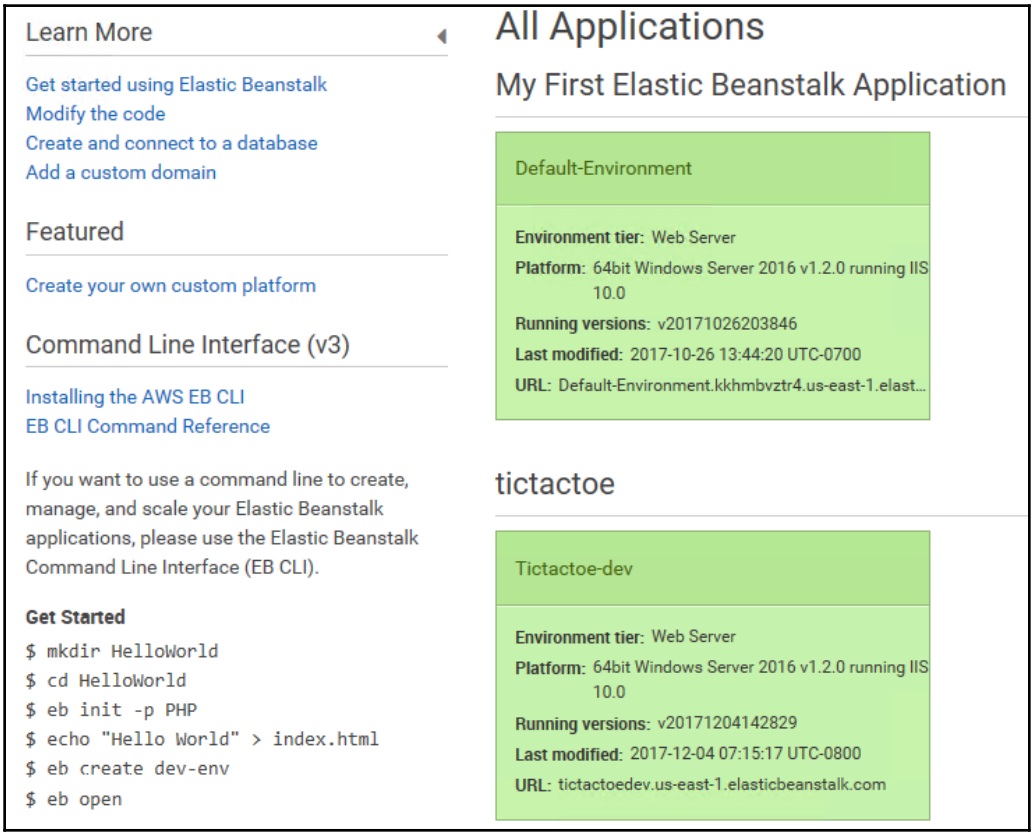
You will now learn how to access logs for applications you have deployed to AWS Elastic Beanstalk, as follows:

1. Publish **Tic-Tac-Toe Web Application** to AWS Elastic Beanstalk. If you do not know how to do this, you can look it up in `Chapter 12`, *Hosting ASP.NET Core 3 Applications*.
2. Start the application, go to **AWS Management Console**, enter `Beanstalk` in the **AWS services Find Services** textbox, and click on the displayed link. You will be redirected to the Elastic Beanstalk welcome page, as follows:

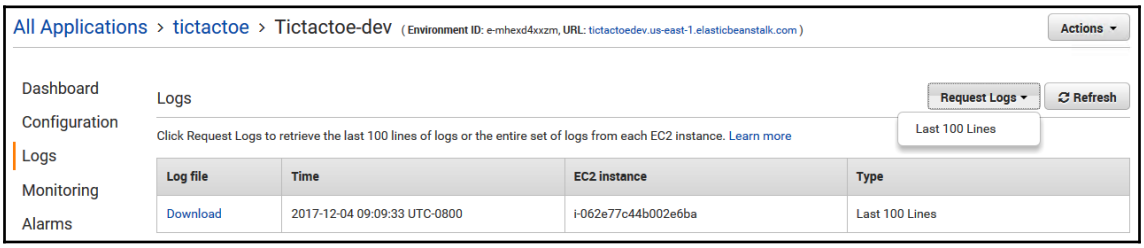


WOW! eBook
www.wowebook.org

3. On the Elastic Beanstalk welcome page, select the `TicTacToe` application you deployed in the preceding step, as shown here:



4. Click on **Logs** in the left-hand menu, and then click on **Request Logs | Last 100 Lines.** You can now download the log files you need to analyze, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

5. Download a log file and check its content, as follows:

```
AWSDeployment.log:

2017-12-04 12:47:04,603 INFO 1 AWSBeanstalkCfnDeployApp.DeployApp - Reading configuration from c:\Program Files\Amazon\ElasticBeanstalk\config\containerconfiguration
2017-12-04 12:47:05,587 INFO 1 AWSBeanstalkCfnDeploy.ContainerConfiguration - Setting SiteName to 'Default Web Site'
2017-12-04 12:47:05,587 INFO 1 AWSBeanstalkCfnDeploy.ContainerConfiguration - Setting AppName to '/'
2017-12-04 12:47:06,649 INFO 1 AWSBeanstalkCfnDeploy.Container - Could not find ElasticBeanstalk/environment section, creating in applicationHost.config
2017-12-04 12:47:08,259 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting directory (Default Web Site\aspnet_client\system_web\2_0_50727).
2017-12-04 12:47:08,259 INFO 1 DeploymentLog - Deleting directory (Default Web Site\aspnet_client\system_web\2_0_50727).
2017-12-04 12:47:08,259 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting directory (Default Web Site\aspnet_client\system_web\4_0_30319).
2017-12-04 12:47:08,259 INFO 1 DeploymentLog - Deleting directory (Default Web Site\aspnet_client\system_web\4_0_30319).
2017-12-04 12:47:08,259 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting directory (Default Web Site\aspnet_client\system_web).
2017-12-04 12:47:08,259 INFO 1 DeploymentLog - Deleting directory (Default Web Site\aspnet_client\system_web).
2017-12-04 12:47:08,259 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting directory (Default Web Site\aspnet_client).
2017-12-04 12:47:08,259 INFO 1 DeploymentLog - Deleting directory (Default Web Site\aspnet_client).
2017-12-04 12:47:08,274 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding directory (Default Web Site\bin).
2017-12-04 12:47:08,274 INFO 1 DeploymentLog - Adding directory (Default Web Site\bin).
2017-12-04 12:47:08,290 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\AWSBeanstalkHelloWorldWebApp.dll).
2017-12-04 12:47:08,290 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\AWSBeanstalkHelloWorldWebApp.dll).
2017-12-04 12:47:08,306 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\AWSSDK.Core.dll).
2017-12-04 12:47:08,306 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\AWSSDK.Core.dll).
2017-12-04 12:47:08,321 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\AWSXRayRecorder.dll).
2017-12-04 12:47:08,321 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\AWSXRayRecorder.dll).
2017-12-04 12:47:08,337 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\log4net.dll).
2017-12-04 12:47:08,337 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\log4net.dll).
2017-12-04 12:47:08,337 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\Newtonsoft.Json.dll).
2017-12-04 12:47:08,337 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\Newtonsoft.Json.dll).
2017-12-04 12:47:08,352 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\System.Net.Http.Formatting.dll).
2017-12-04 12:47:08,352 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\System.Net.Http.Formatting.dll).
2017-12-04 12:47:08,368 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\System.Web.Http.dll).
2017-12-04 12:47:08,368 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\System.Web.Http.dll).
2017-12-04 12:47:08,399 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\bin\System.Web.Http.WebHost.dll).
2017-12-04 12:47:08,399 INFO 1 DeploymentLog - Adding file (Default Web Site\bin\System.Web.Http.WebHost.dll).
2017-12-04 12:47:08,415 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\Default.aspx).
2017-12-04 12:47:08,415 INFO 1 DeploymentLog - Adding file (Default Web Site\Default.aspx).
2017-12-04 12:47:08,415 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Adding file (Default Web Site\Global.asax).
2017-12-04 12:47:08,415 INFO 1 DeploymentLog - Adding file (Default Web Site\Global.asax).
2017-12-04 12:47:08,415 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting file (Default Web Site\iisstart.htm).
2017-12-04 12:47:08,415 INFO 1 DeploymentLog - Deleting file (Default Web Site\iisstart.htm).
2017-12-04 12:47:08,415 INFO 1 AWSBeanstalkCfnDeploy.DeploymentUtils - Deleting file (Default Web Site\iisstart.png).
```

You have seen how to handle logging in various environments, on-premises, and in the cloud. The next section will introduce you to monitoring, and how it can aid you in analyzing problems in real time.

Monitoring ASP.NET Core 3 applications

In the previous section, you saw how to generate and analyze application logs for your ASP.NET Core 3 web applications, which will help you better understand unexpected behavior and application bugs. This will help IT operations to trace the different steps after an event has occurred until the root cause of a problem has been found.

However, it will not help them to constantly monitor and supervise applications, since using logging mechanisms, in this case, will result in bad performance and negative overall impact on the application. Logging is not the right solution for continuous monitoring!

The goal of monitoring is to analyze and supervise a large number of application metrics in real time, and to automatically detect application anomalies. The metrics need to have a very low message footprint for this to work efficiently.

WOW! eBook
www.wowebook.org

The most commonly known monitoring frameworks for ASP.NET Core 3 are listed here:

- `EventSource` with ETW, which is very fast, and strongly typed. This was introduced with .NET 4 and works only on Windows
- `DiagnosticSource`, which is very similar to `EventSource`, works cross-platform, like `EventSource` with ETW for Windows, and like LTTng for Linux

For more information on ETW, go to the following website: `https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing`.
For more information on LTTng, go to the following website: `http://lttng.org`.

On top of these frameworks, most public cloud providers supply their own monitoring solutions. For Microsoft Azure, it is recommended to use Azure Application Insights, for instance, while you should use CloudWatch for AWS. These two monitoring solutions are fully SaaS and are much more integrated with the respective public cloud provider portals.

Monitoring on-premises and in Docker

There are no standard monitoring solutions for on-premises and Docker environments as such, but there are some community-approved monitoring frameworks, such as `EventSource` or `DiagnosticSource`, which you can use to implement your own solutions.

Since these frameworks respect market standards such as ETW, IT operations will be able to connect your ASP.NET Core 3 web applications using their standard monitoring tools, and they will like that very much!

An example would be PerfMon on Windows, which can receive ETW events and generate diagrams for monitoring purposes.

While using `DiagnosticSource`, you start by creating a listener. This listener receives application events and provides event names and parameters. The easiest way to create a listener is to create a **Plain Old CLR Objects** (**POCO**) class, which contains methods that need to be decorated with the `[DiagnosticName]` decorator and is designed to accept parameters of the appropriate types.

WOW! eBook
www.wowebook.org

The following example explains how to use `DiagnosticSource` to add monitoring to your ASP.NET Core 3 applications in on-premises and Docker environments:

1. Open the Tic-Tac-Toe web project in Visual Studio 2019, and add a new folder called `Monitoring`; in this folder, add a new class called `ApplicationDiagnosticListener`, as follows:

```
public class ApplicationDiagnosticListener
{
  [DiagnosticName("TicTacToe.MiddlewareStarting")]
  public virtual void OnMiddlewareStarting(HttpContext
    httpContext)
  {
    Console.WriteLine
        ($"TicTacToe Middleware Starting, path:
         {httpContext.Request.Path}");
  }

  [DiagnosticName("TicTacToe.NewUserRegistration")]
  public virtual void NewUserRegistration(string name)
  {
    Console.WriteLine($"New User Registration {name}");
  }
}
```

2. Update the `Configure` method in the `Startup` class, add `DiagnosticListener`, and subscribe to `ApplicationDiagnosticListener`, as shown here:

```
public void Configure(IApplicationBuilder app,
 IHostingEnvironment env, DiagnosticListener
    diagnosticListener)
{
  var listener = new ApplicationDiagnosticListener();
  diagnosticListener.SubscribeWithAdapter(listener);
  ...
}
```

3. Update `CommunicationMiddleware`, add a new private member called `_diagnosticSource`, and update the constructor, as follows:

```
private readonly RequestDelegate _next;
private DiagnosticSource _diagnosticSource;
public CommunicationMiddleware(RequestDelegate next,
 DiagnosticSource diagnosticSource)
{
  _next = next;
```

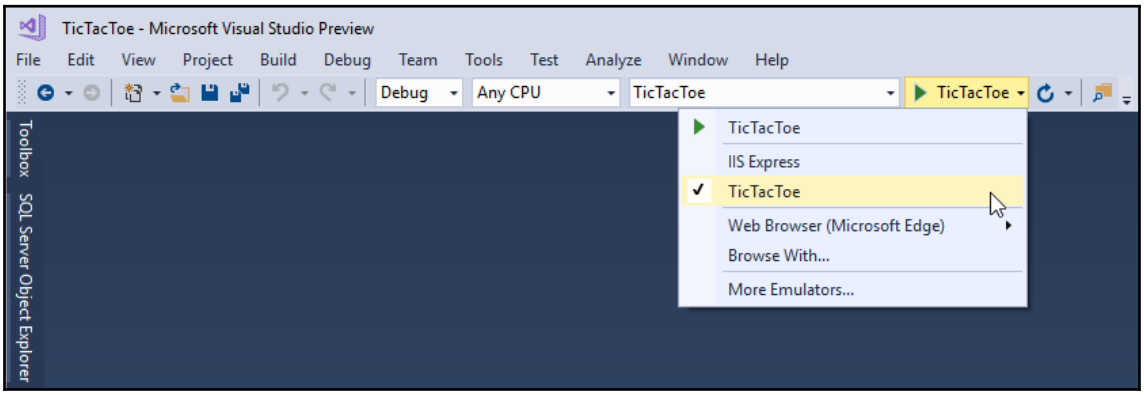

WOW! eBook
www.wowebook.org

```
    _diagnosticSource = diagnosticSource;
  }
```

4. Update the `Invoke` method in `CommunicationMiddleware`, and write an event if the diagnostic source is enabled, as follows:

```
public async Task Invoke(HttpContext context)
{
  if (context.WebSockets.IsWebSocketRequest)
  {
    if (_diagnosticSource.IsEnabled
       ("TicTacToe.MiddlewareStarting"))
    {
      _diagnosticSource.Write("TicTacToe.
        MiddlewareStarting",
      new
      {
        httpContext = context
      });
    }
    ...
```

5. Change the debugging settings in Visual Studio 2019, and set the project and emulator to **TicTacToe**, as follows:



WOW! eBook
www.wowebook.org

6. Start the application in Debug mode by pressing *F5*. A console will be opened automatically. Register a new user and check the console output; you will see the **TicTacToe Middleware Starting** message, as shown here:

```
TicTacToe
      Executed action TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe) in 2657.7493ms
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TicTacToe.Controllers.UserRegistrationController.EmailConfirmation (TicTacToe) in 2657.7493ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2744.5098ms 200 text/html; charset=utf-8
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2744.5098ms 200 text/html; charset=utf-8
infoinfo: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:59929/CheckEmailConfirmationStatus
: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:59929/CheckEmailConfirmationStatus
infoinfo: Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationHandler[8]
      AuthenticationScheme: Cookies was successfully authenticated.
: Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationHandler[8]
      AuthenticationScheme: Cookies was successfully authenticated.
TicTacToe Middleware Starting, path: /CheckEmailConfirmationStatus
```

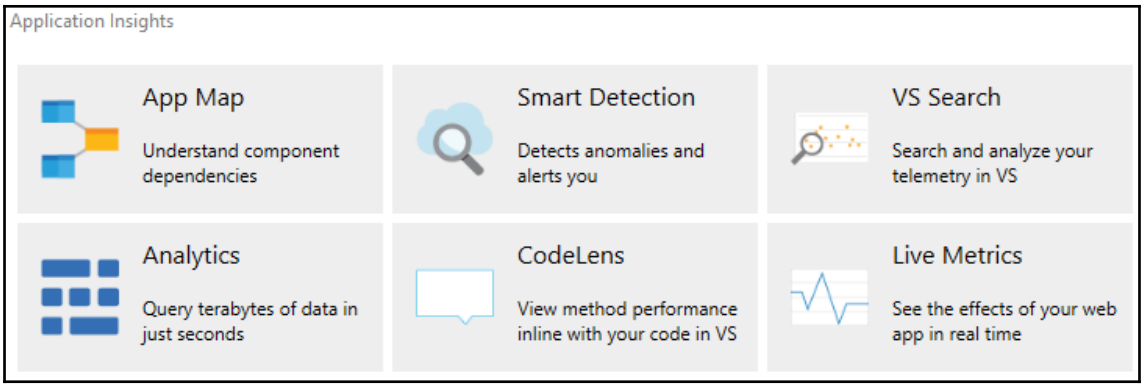
As already mentioned, sending logging and monitoring data to the console is a possible solution for on-premises environments, and is a recommended solution for Docker environments.

Monitoring in Microsoft Azure

Microsoft Azure provides an integrated solution called Azure Application Insights, which allows IT operations to monitor applications, resources, and services in real time. It works for the whole Azure subscription and includes dashboards and diagrams for quick access to analytic data.

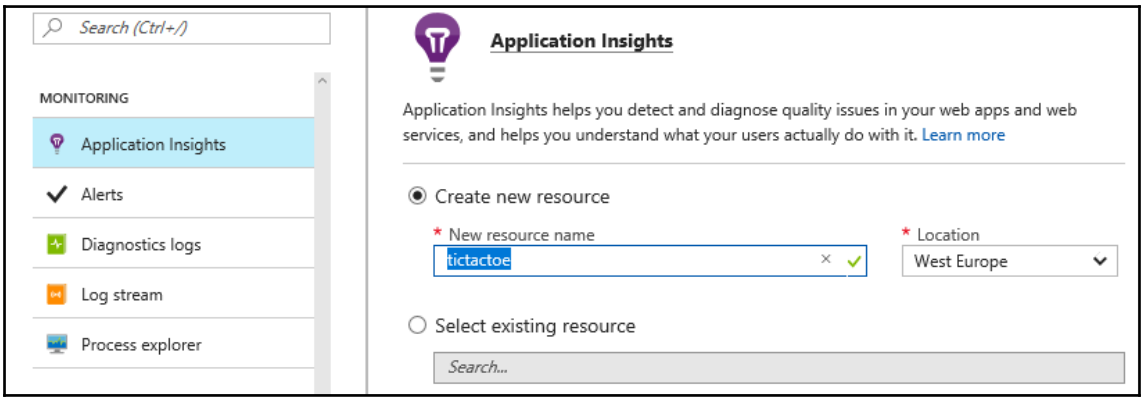
The following diagram illustrates some features of Azure Application Insights:

WOW! eBook
www.wowebook.org



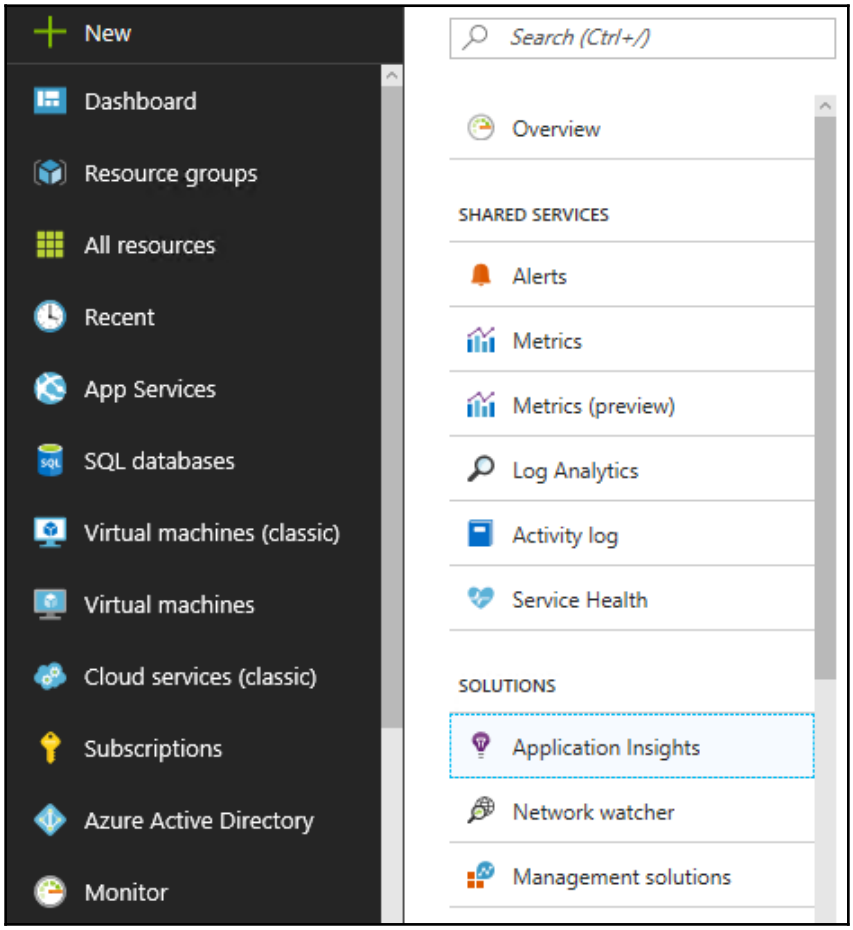
Let's use Application Insights in an easy-to-understand example; for that, you will start by creating a new Azure Application Insights resource in Microsoft Azure with its corresponding API key, as follows:

1. Go to the Microsoft Azure portal website, click on **App Services** in the menu, select the **Tic-Tac-Toe App Service** you have deployed and configured in the preceding example, scroll down until you see the **MONITORING** section, click on **Application Insights**, fill out all the fields, and then click on the **OK** button. A new **Application Insights** resource will be created for you, as follows:



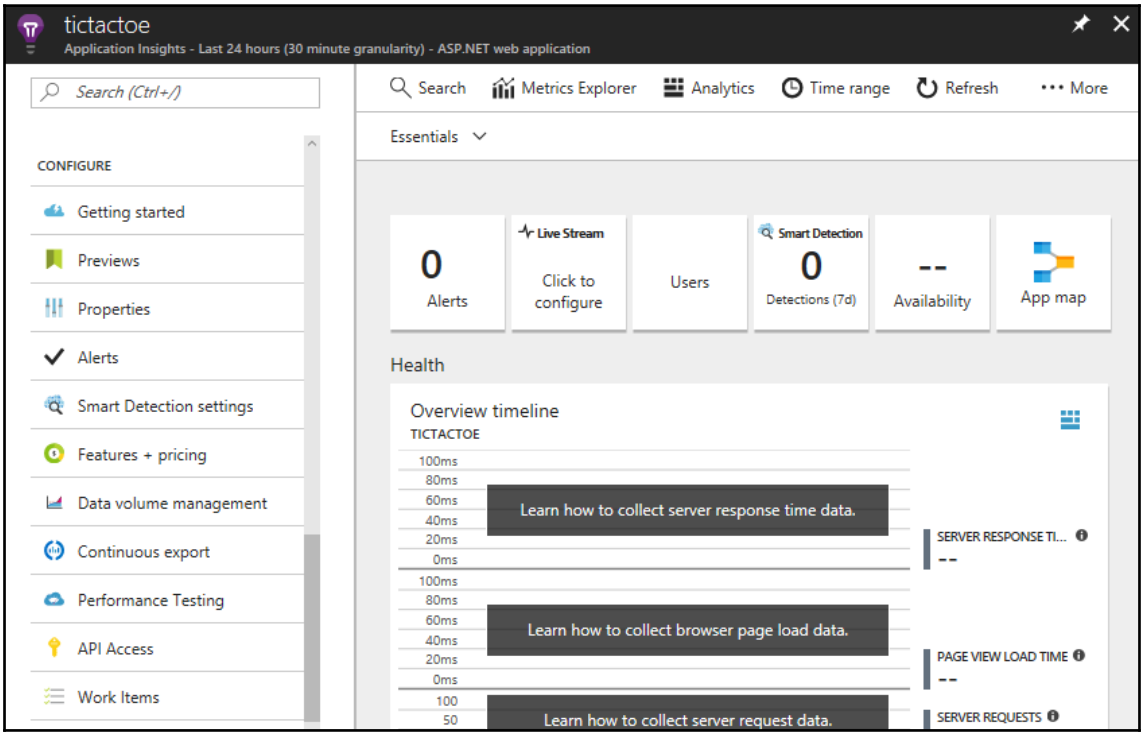
WOW! eBook
www.wowebook.org

2. Click on **Monitor** in the menu. A new tab will be displayed. Go to the **SOLUTIONS** section and choose **Application Insights**, and then select the created **Application Insights** resource, as follows:



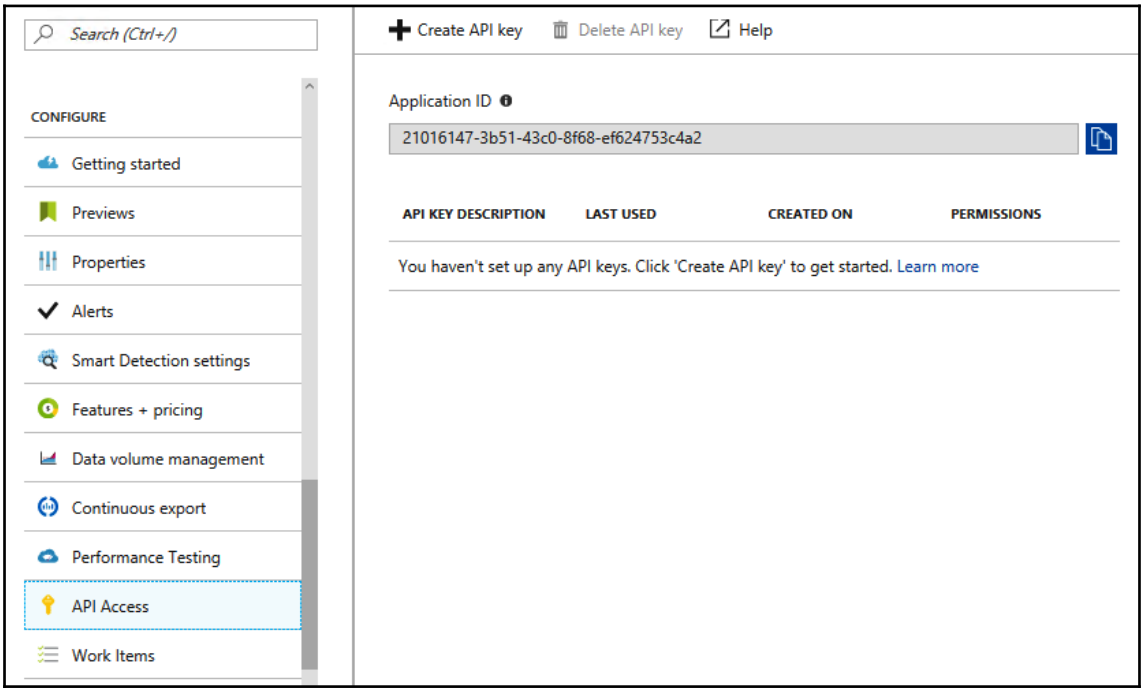
WOW! eBook
www.wowebook.org

3. The **Application Insights** resource tab will be displayed; scroll down until you see the **Configure** section, and then click on **API Access**, as shown in the following screenshot:



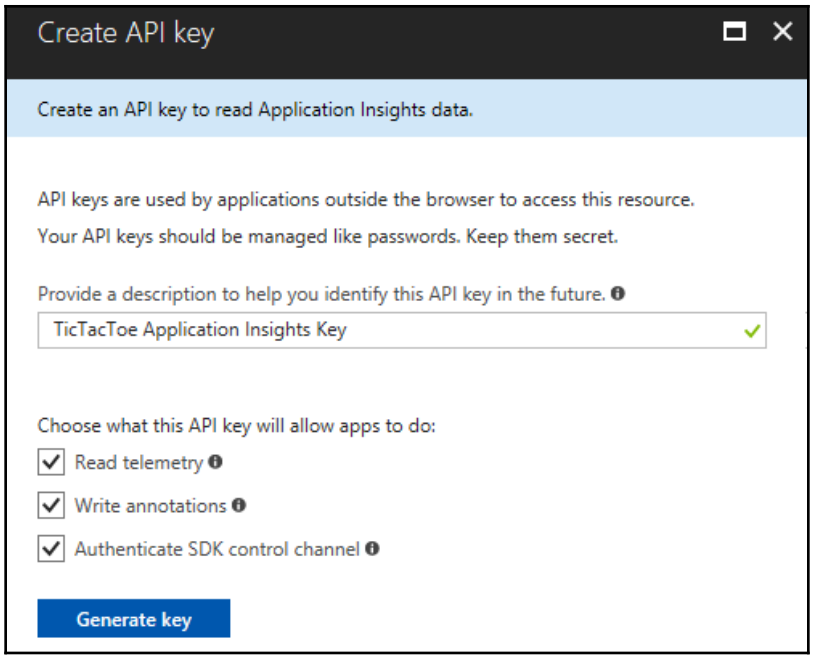
WOW! eBook
www.wowebook.org

4. Click on **Create API key** to be able to generate a key, which will be used for the Tic-Tac-Toe sample application, as shown in the following screenshot:



5. Configure the API key access rights (**Read telemetry**, **Write annotations**, **Authenticate SDK control channel**) and give it a meaningful name, as shown in the following screenshot:

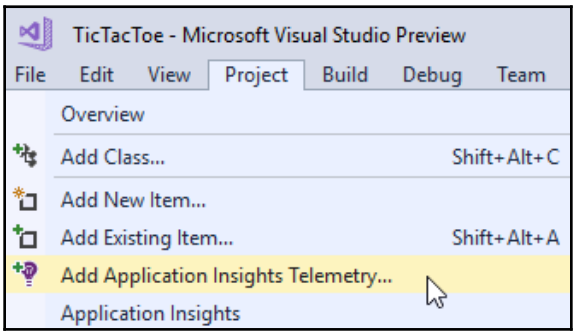
WOW! eBook
www.wowebook.org



You have now finished the creation and configuration of the Application Insights resource in Microsoft Azure. Visual Studio 2019 contains some advanced built-in features that will allow you to connect your ASP.NET Core 3 application directly from within the **integrated development environment** (**IDE**).

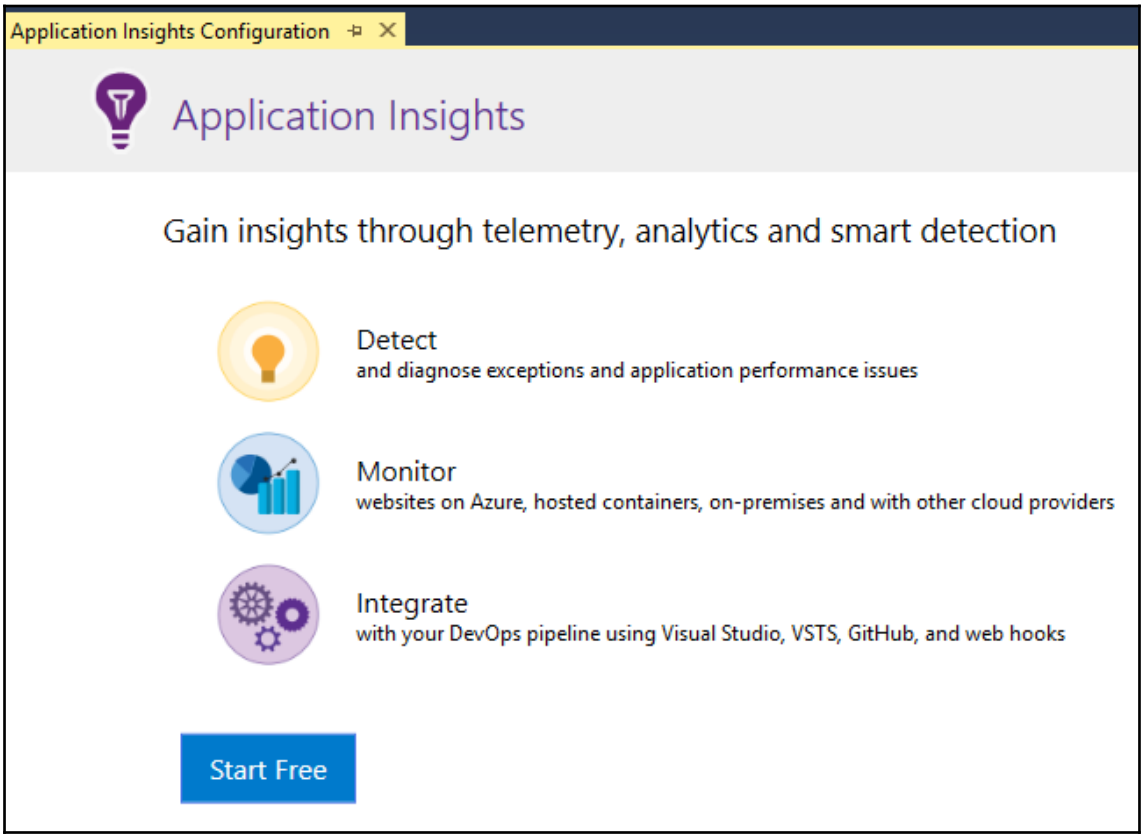
In the next steps, you will configure the ASP.NET Core 3 web application for Azure Application Insights as follows:

1. Open the Tic-Tac-Toe web project, click on **Project** in the top menu, and select **Add Application Insights Telemetry...**, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

2. The **Application Insights Configuration** page will be displayed. Click on the **Start Free** button, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

3. Enter your account and subscription details, select a resource, and click on the **Register** button, as shown in the following screenshot:

Resource

tictactoe (Existing resource)

Configure settings...

Base Monthly Price	Free
Included Data	1 GB / Month
Additional Data	$2.30 per GB*
Data retention (raw and aggregated data)	90 days

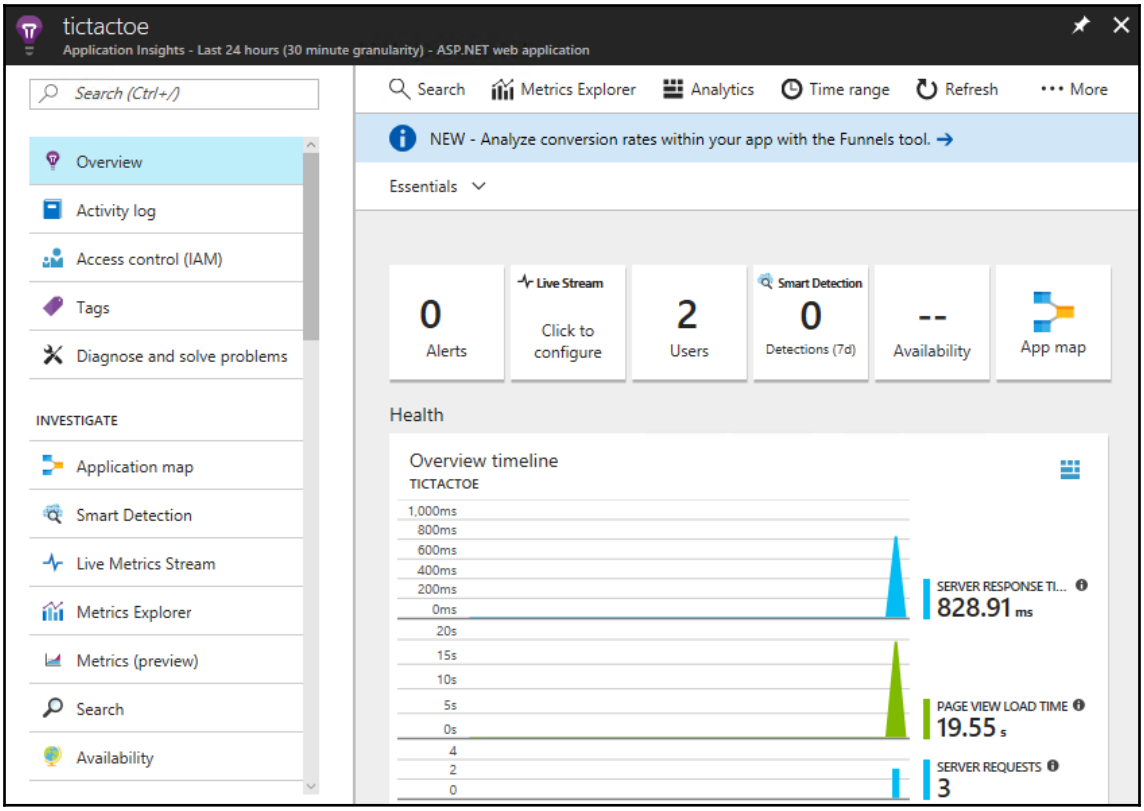
*Pricing is subject to change. Visit our pricing page for most recent pricing details.

Register

4. Republish the Tic-Tac-Toe web application to the Microsoft Azure App Service so that the Application Insights configurations are applied.
5. Go to the Microsoft Azure portal website, click on **Monitor** in the menu, scroll down to the **Solutions** section and click on **Application Insights**, and then select the newly created **Application Insights** resource.

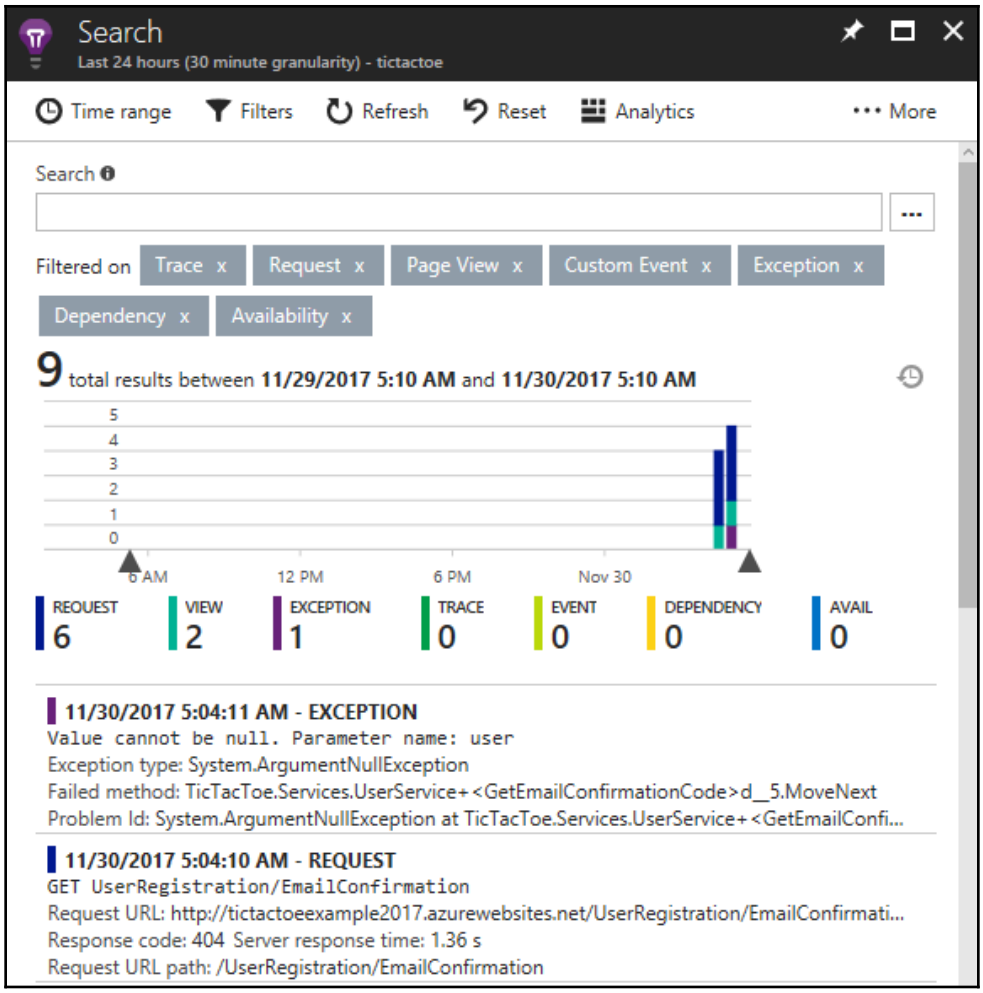
WOW! eBook
www.wowebook.org

6. The **Application Insights** dashboard will be displayed. It serves to get a global overview, as well as to dive deeper into the different monitoring areas, as shown in the following screenshot:



7. Click on **Search** to see the application flow; here, you can see that an error has occurred during the user registration process, as follows:

WOW! eBook
www.wowebook.org



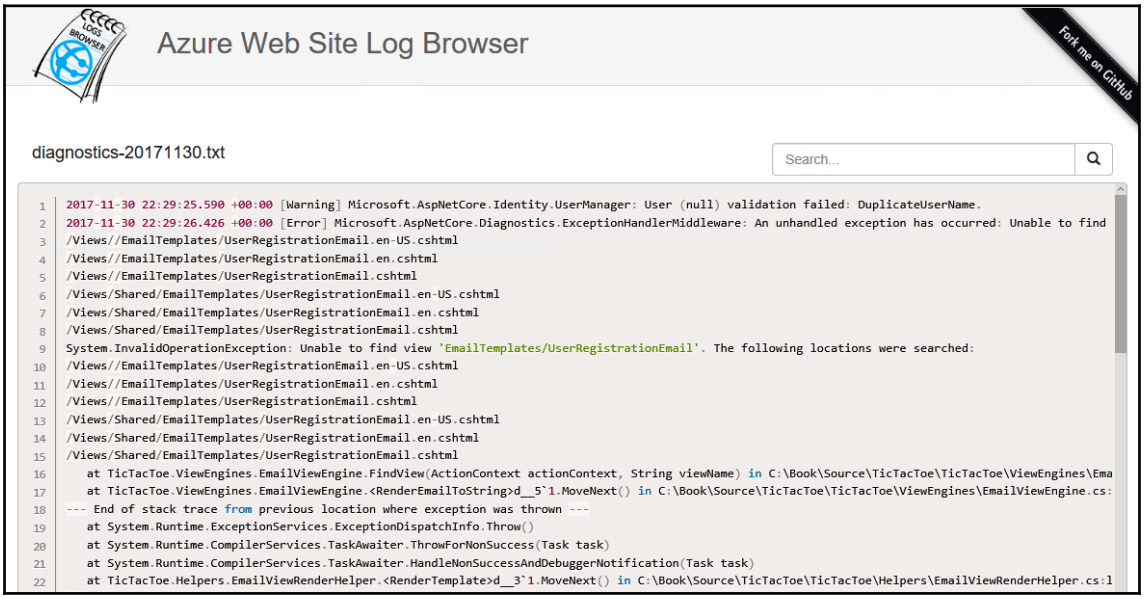
You may have already seen these errors in `Chapter 12`, *Hosting ASP.NET Core 3 Applications*, after having deployed the Tic-Tac-Toe application to either Microsoft Azure or AWS, as well as in the preceding logging section in this chapter. Everything is working locally and in Docker, but when you deploy it to the public cloud, it is not working anymore. Very strange! We cannot wait any longer; it really needs to be fixed!

We will now analyze the problem in more detail, and try to understand what needs to be done to solve it, as follows:

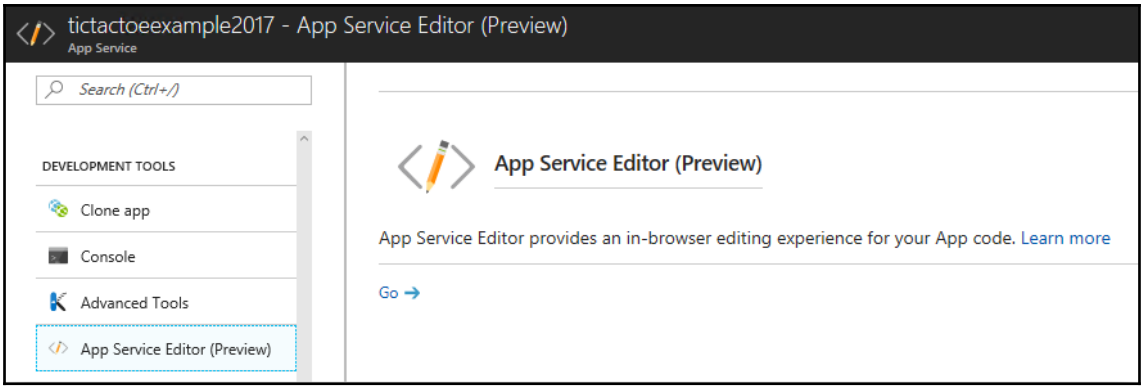
1. In Azure Application Insights, you can clearly see that there is a problem with the user registration: more specifically, a 404 Not Found HTTP response.

WOW! eBook
www.wowebook.org

2. When looking into the log file, as explained in the preceding section, you can see that the `UserRegistrationEmail` view in the `EmailTemplates` folder cannot be found, which then leads to additional errors, as shown in the following screenshot:

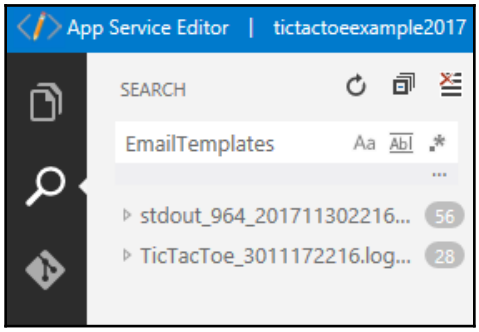


3. Go to the Microsoft Azure portal website, click on **App Services** in the menu, select the **Tic-Tac-Toe App Service** you have deployed and configured in the preceding example, scroll down until you see the **DEVELOPMENT TOOLS** section, click on **App Service Editor (Preview)**, and then click on the **Go** link, as shown in the following screenshot:



WOW! eBook
www.wowebook.org

4. A new window with the **App Service Editor** page will automatically be opened; click on the **SEARCH** button and search for the `EmailTemplates` folder. It cannot be found because all views were precompiled into a single **dynamic-link library** (**DLL**) called `TicTacToe.PrecompiledViews.dll` during the publishing process, as follows:



5. Apply a temporary fix for this problem by deactivating the precompilation during the publishing process. Open the `.csproj` file of the Tic-Tac-Toe web project, and then add the following configuration elements to the `PropertyGroup` section:

```
        <PropertyGroup>
          ...
          <PreserveCompilationContext>true
           </PreserveCompilationContext>
<MvcRazorCompileOnPublish>false</MvcRazorCompileOnPublish>
        </PropertyGroup>
```

Note that this is only a temporary fix, for example purposes. You should reactivate precompilation, and target the precompiled views in your code for a more industrialized and production-ready solution.

6. Republish the Tic-Tac-Toe web application to the Microsoft Azure App Service. Everything should now be working, including the user registration.

WOW! eBook
www.wowebook.org

Note that you have to register a completely new user with a strong password such as `Azerty1234!`, for example, otherwise you might get additional errors if you don't. The application is missing some more advanced error handling due to a lack of space within the book. Keep in mind that it was only given to better understand all the ASP.NET Core 3 concepts. You can, however, use the sample application as a base and then refine it as you like, and add the missing error handling.

You have seen how to configure your ASP.NET Core 3 web applications, and are able to monitor them by using Azure Application Insights. You have even identified a problem during the user registration of the application. You have analyzed the logging and monitoring data, and you were able to solve the problem.

This works exceptionally well with .NET Core code, but, for now, you cannot see whether any errors occur in the JavaScript parts of your applications. Since modern applications include a large number of JavaScript code, it would be great if you were able to monitor these parts also; right? Well, you can do that; you just have to adapt the code a little bit.

Let's see how to adapt the code and be able to monitor JavaScript application flows, as follows:

1. Start Visual Studio 2019 and open the Tic-Tac-Toe web project, update the `_ViewImports.cshtml` file in the `Views` folder, and add the Application Insights JavaScript snippet to the bottom of the file, as follows:

```
@inject Microsoft.ApplicationInsights.AspNetCore
 .JavaScriptSnippet JavaScriptSnippet
```

2. Update the layout page and mobile layout page, and then add the following line to the `head` section of the two pages:

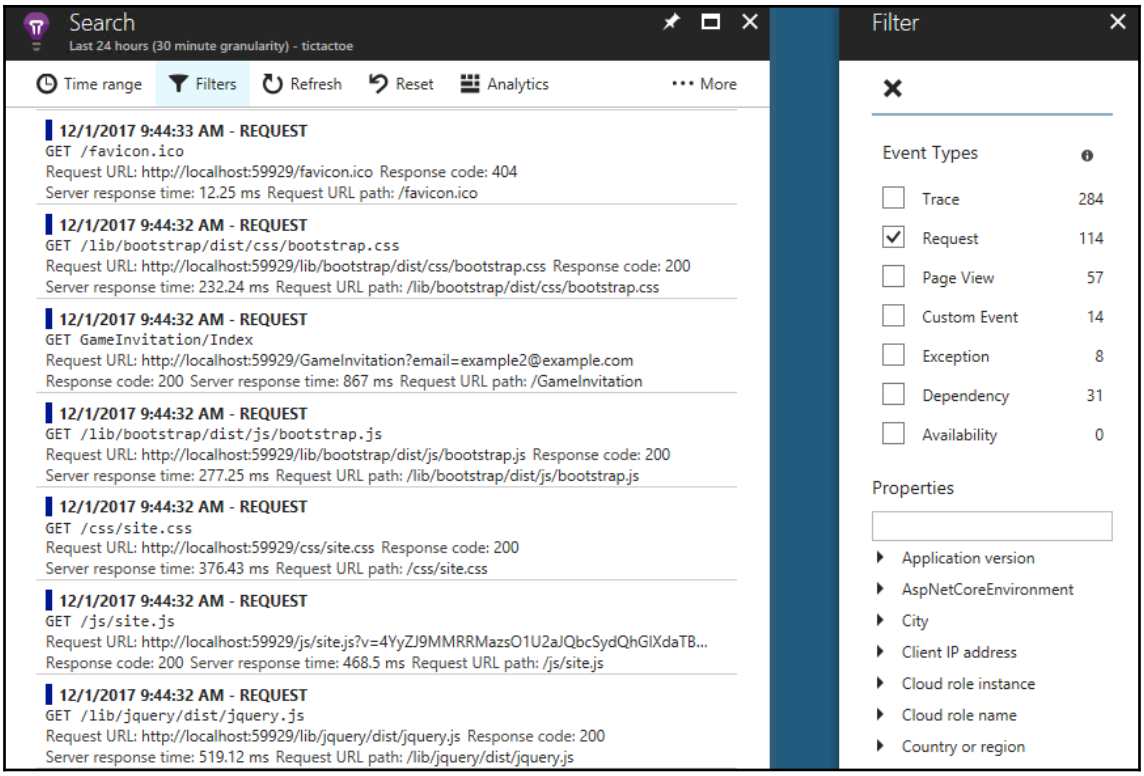
```
@Html.Raw(JavaScriptSnippet.FullScript)
```

3. Update the `Startup` class, and register the Application Insights service, as follows:

```
services.AddApplicationInsightsTelemetry(_configuration);
```

4. Republish the Tic-Tac-Toe web application to the Microsoft Azure App Service so that the new Application Insights configuration is applied.
5. Start the application and open the **Application Insights** dashboard in the Microsoft Azure portal website, click on **Search**, and then click on **Filters** and select **Request** only, deselecting all the other event types, as shown in the following screenshot:

WOW! eBook
www.wowebook.org



Great! You are able to constantly monitor your entire application, whether it be on the JavaScript side or on the .NET Core side, which will turn out to be quite useful in the case of incorrect behavior.

In the last step, you will learn how to add and monitor custom metrics, which will allow you to trace business metrics in your applications, as follows:

1. Open the Tic-Tac-Toe web project, and add a new service named `AzureApplicationInsightsMonitoringService` to the `Services` folder, as follows:

```
public class AzureApplicationInsightMonitoringService
{
  readonly TelemetryClient _telemetryClient =  new
    TelemetryClient();

  public void TrackEvent(string eventName, TimeSpan
  elapsed,
   IDictionary<string, string> properties = null)
```


WOW! eBook
www.wowebook.org

```
    {
      var telemetry = new EventTelemetry(eventName);
      telemetry.Metrics.Add("Elapsed",
       elapsed.TotalMilliseconds);

      if (properties != null)
        foreach (var property in properties)
        {
          telemetry.Properties.Add(property.Key,
           property.Value);
        }
      _telemetryClient.TrackEvent(telemetry);
    }
  }
```

2. Extract the interface from the Azure `ApplicationInsightsMonitoringService` class and call it `IMonitoringService`.
3. Add a new option called `MonitoringOptions` to the `Options` folder, as follows:

```
public class MonitoringOptions
{
  public string MonitoringType { get; set; }
  public string MonitoringSetting { get; set; }
}
```

4. Update the `Configure` method in the `Startup` class, and register the Azure `ApplicationInsightsMonitoringService` class if it has been configured in the `appsettings.json` configuration file, as follows:

```
...
services.AddApplicationInsightsTelemetry(_configuration);
var section = _configuration.GetSection("Monitoring");
var monitoringOptions = new MonitoringOptions();
section.Bind(monitoringOptions);
services.AddSingleton(monitoringOptions);

if (monitoringOptions.MonitoringType ==
  "azureapplicationinsights")
{
  services.AddSingleton<IMonitoringService,
    AzureApplicationInsightsMonitoringService>();
}
```


WOW! eBook
www.wowebook.org

5. Update the `UserService` and add a new private member called `_telemetryClient`, and then update the constructor to initialize the private member, as follows:

```
...
private readonly IMonitoringService _telemetryClient;
public UserService(RoleManager<RoleModel> roleManager,
 ApplicationUserManager userManager, ILogger<UserService>
 logger, SignInManager<UserModel>
   signInManager,IMonitoringService telemetryClient)
{
  ...
  _telemetryClient = telemetryClient;
  ...
}
```

6. Update the `RegisterUser` method in `UserService` to use the `TrackEvent` method, and then add a custom metric called `RegisterUser`, as follows:

```
...
finally
{
  stopwatch.Stop();
  _telemetryClient.TrackEvent("RegisterUser",
    stopwatch.Elapsed);
  _logger.LogTrace($"Start register user {userModel.Email}
   finished at {DateTime.Now} - elapsed
   {stopwatch.Elapsed.TotalSeconds} second(s)");
}
...
```

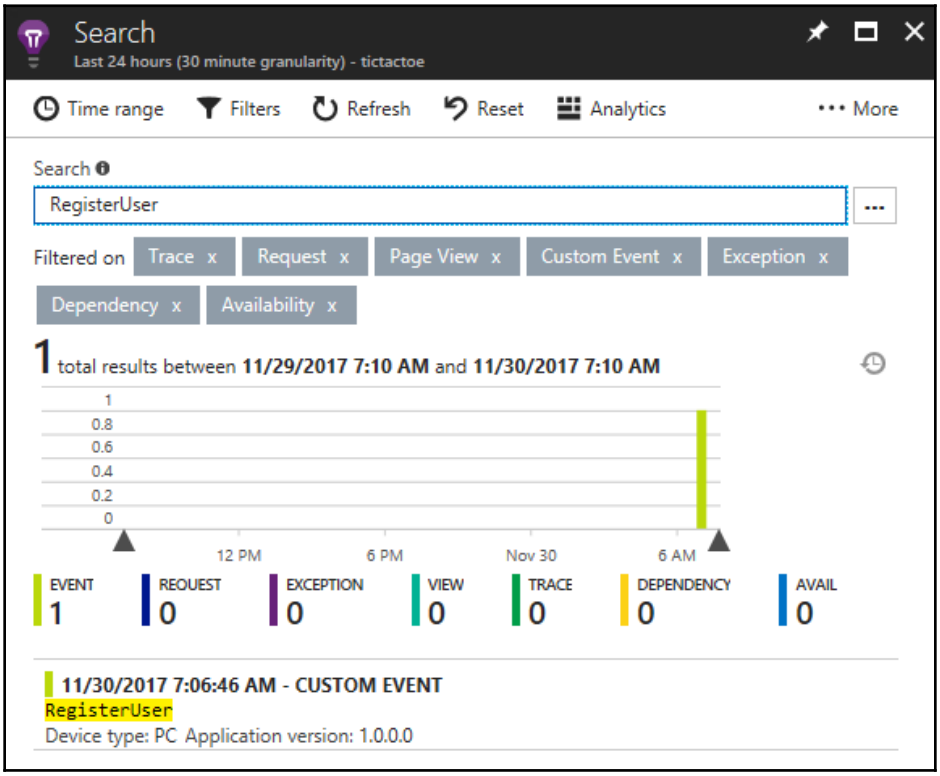
7. Update the `appsettings.json` configuration file, add a new `Monitoring` section, and then configure it for Azure Application Insights, as follows:

```
"Monitoring": {
  "MonitoringType": "azureapplicationinsights",
  "MonitoringSettings": ""
}
```

8. Republish the Tic-Tac-Toe web application to the Microsoft Azure App Service so that the new Application Insights configurations are applied.

WOW! eBook
www.wowebook.org

9. Start the application and open the **Application Insights** dashboard on the Microsoft Azure portal website, click on **Search**, and enter `RegisterUser` as a search term; you will only see the custom `RegisterUser` business metric now, as follows:



That is all we need in order to monitor even reasonably complex applications on Azure, and if you prefer to host your applications with AWS, the next section will show you how we can achieve similar functionality in the AWS platform.

Monitoring in AWS

Just like Microsoft Azure, AWS provides an integrated solution, which allows IT operations to monitor applications, resources, and services in real time. In AWS, this solution is called CloudWatch. It provides nearly the same features as Application Insights, meaning it works for the entire AWS subscription, and includes dashboards and diagrams, for quick access to analytic data.

WOW! eBook
www.wowebook.org

The following example illustrates how to use AWS CloudWatch to monitor generic metrics and custom metrics, so that you can learn how to deploy it for your own needs:

1. Open the Tic-Tac-Toe web project, and download and install the **Amazon Web Services SDK for .NET - Core Runtime** NuGet package called `AWSSDK.Core`, as well as the **Amazon Web Services CloudWatch** NuGet package called `AWSSDK.CloudWatch`.
2. Add a new service called `AmazonWebServicesMonitoringService` to the `Services` folder, make it inherit the `IMonitoringService` interface, and implement the `TrackEvent` method with the AWS-specific code, as shown in the following code block:

```
public class AmazonWebServicesMonitoringService :
 IMonitoringService
{
  readonly AmazonCloudWatchClient _telemetryClient = new
    AmazonCloudWatchClient();
  public void TrackEvent(string eventName, TimeSpan
   elapsed,
   IDictionary<string, string> properties = null)
  {
    ...
  }
}
```

3. Here is the actual code in the `TrackEvent` method:

```
var dimension = new Dimension { Name = eventName, Value = eventName
};
var metric1 = new MetricDatum
{
  Dimensions = new List<Dimension> { dimension },
  MetricName = eventName, StatisticValues = new StatisticSet(),
  Timestamp = DateTime.Today, Unit = StandardUnit.Count
};

if (properties?.ContainsKey("value") == true)
  metric1.Value = long.Parse(properties["value"]);
else   metric1.Value = 1;

var request = new PutMetricDataRequest
{ MetricData = new List<MetricDatum>() { metric1 }, Namespace =
eventName  };
 _telemetryClient.PutMetricDataAsync(request).Wait();
```


WOW! eBook
www.wowebook.org

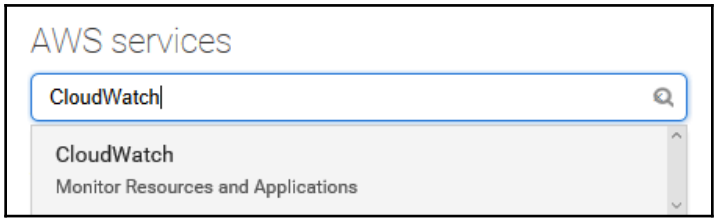
4. Update the `Configure` method in the `Startup` class, and register the **Amazon Web Services Cloud Watch Monitoring Service**, if it has been configured in the `appsettings.json` configuration file, as follows:

```
...
if (monitoringOptions.MonitoringType ==
 "azureapplicationinsights")
{
  services.AddSingleton<IMonitoringService,
   AzureApplicationInsightsMonitoringService>();
}
else if (monitoringOptions.MonitoringType ==
 "amazonwebservicescloudwatch")
{
  services.AddSingleton<IMonitoringService,
   AmazonWebServicesMonitoringService>();
}
```

5. Update the `Monitoring` section in the `appsettings.json` configuration file, and configure it for AWS CloudWatch, as follows:

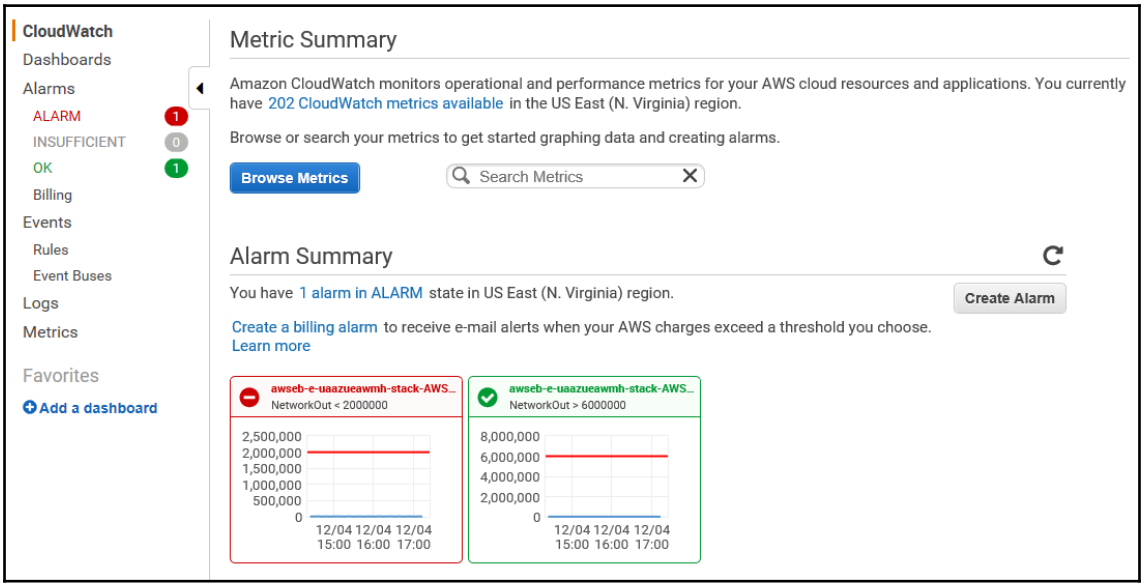
```
"Monitoring": {
  "MonitoringType": "amazonwebservicescloudwatch",
  "MonitoringSettings": ""
}
```

6. Publish the Tic-Tac-Toe web application to AWS Elastic Beanstalk, so that the new AWS CloudWatch configurations are applied. If you do not know how to do this, you can look it up in `Chapter 12`, *Hosting ASP.NET Core 3 Applications*.
7. Start the application. Go to the **AWS Management Console**, enter `CloudWatch` in the **AWS services** textbox, and click on the displayed link. You will be redirected to the AWS CloudWatch welcome page, as follows:

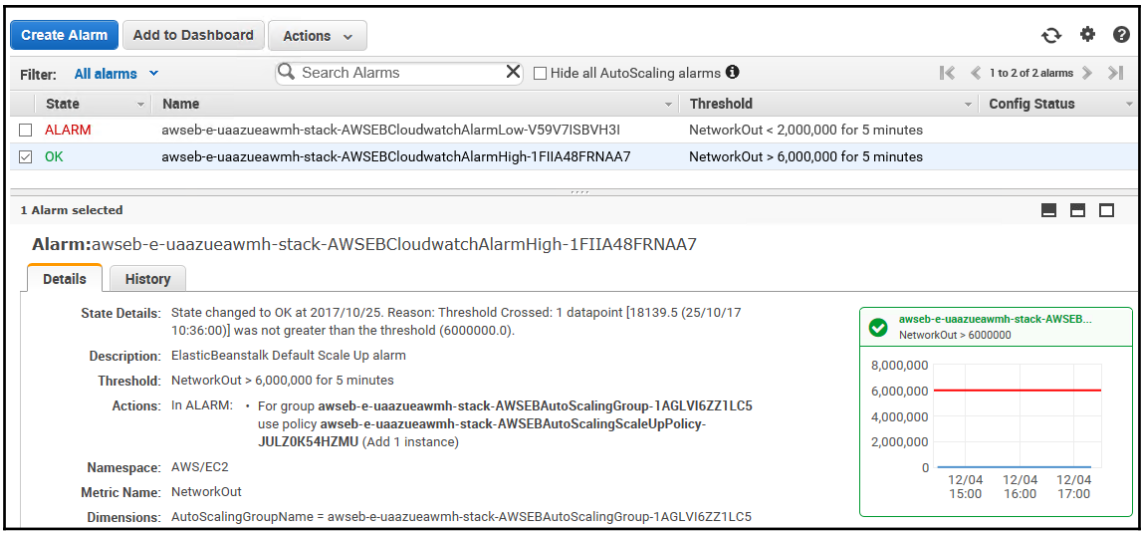


WOW! eBook
www.wowebook.org

8. On the **CloudWatch** welcome page, click on the `TicTacToe` application, as follows:

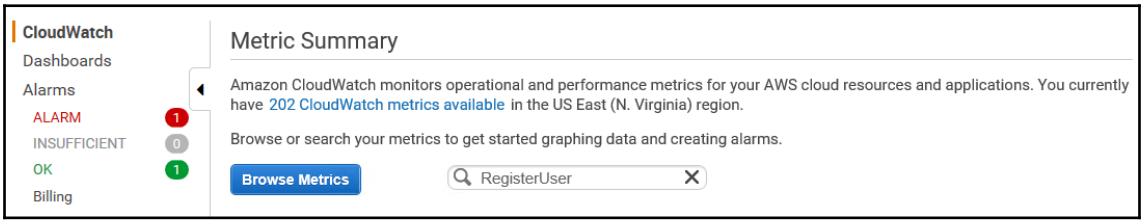


9. Click on an alarm to get more specific details about it, as follows:

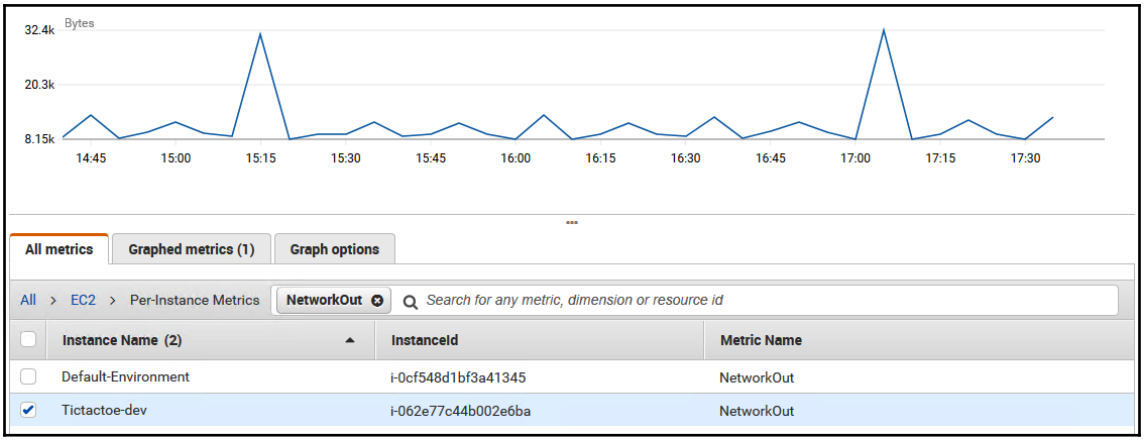


WOW! eBook
www.wowebook.org

10. Return to the **CloudWatch** welcome page, enter `RegisterUser` as a search term in the textbox, and then click on **Browse Metrics**, as follows:



11. You will see a diagram, as shown here, with the custom `RegisterUser` business metric:



This should suffice to give you a feel of how you can monitor your platform, but it is advised that you play around to see all the extra capabilities. I'm quite certain that you will have fun detecting and preventing anomalies in whatever application for which you are responsible.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we discussed how to manage and supervise your ASP.NET Core web applications to help IT operations better understand what is happening during runtime, before and after errors occur.

We talked about the concept of logging, and how it can help reduce the time to understand and fix bugs. We illustrated different logging solutions: on-premises, in Microsoft Azure, in AWS, and in Docker.

You experienced how to configure logging in a Microsoft Azure environment using Azure App Service and Azure App Service Diagnostics, as well as the **Azure Web Site Log Browser** extension for log file analysis, in a detailed example.

You then saw how to do the same in AWS by accessing and downloading application logs, using AWS CloudWatch.

We then introduced the concepts of monitoring, and explained how to add monitoring to on-premises and Docker environments.

You configured Azure Application Insights to monitor your ASP.NET Core web applications in real time. You were even able to understand and solve the mystery behind the `404 Not Found` problem.

In the last step, we showed you how to work with monitoring in an AWS environment, using AWS CloudWatch.

In the next chapter, we will…well, there is no next chapter. You have seen everything this book has to offer. We hope that you liked it and that you have found some value in understanding and assimilating the numerous examples we have given.

It is now up to you to create your own experiences, and to further improve your ASP.NET Core skills.

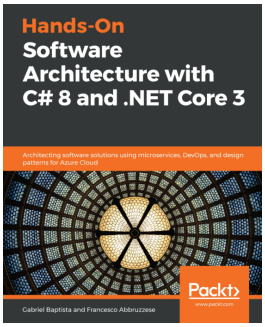
You can now start your journey as a veteran, as Nicolas Clerc (Cloud Architect, Microsoft France) has stated in his *Foreword* at the beginning of this book.

Good luck with that, and thank you for having taken the time to read the different chapters, and for having stayed with us for so long!

WOW! eBook
www.wowebook.org

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

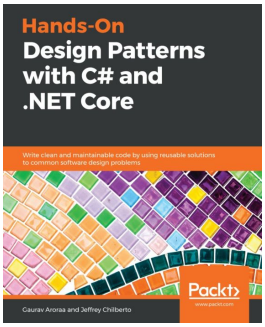


Hands-On Software Architecture with C# 8 and .NET Core 3
Gabriel Baptista, Francesco Abbruzzese

ISBN: 978-1-78980-093-7

- Overcome real-world architectural challenges and solve design consideration issues
- Apply architectural approaches like Layered Architecture, service-oriented architecture (SOA), and microservices
- Learn to use tools like containers, Docker, and Kubernetes to manage microservices
- Get up to speed with Azure Cosmos DB for delivering multi-continental solutions
- Learn how to program and maintain Azure Functions using C#
- Understand when to use test-driven development (TDD) as an approach for software development
- Write automated functional test cases for your projects

WOW! eBook
www.wowebook.org



Hands-On Design Patterns with C# and .NET Core
Gaurav Aroraa, Jeffrey Chilberto

ISBN: 978-1-78913-364-6

- Make your code more flexible by applying SOLID principles
- Follow the test-driven development (TDD) approach in your .NET Core projects
- Get to grips with efficient database migration, data persistence, and testing techniques
- Convert a console application to a web application using the right MVP
- Write asynchronous, multithreaded, and parallel code
- Implement MVVM and work with RxJS and AngularJS to deal with changes in databases
- Explore the features of microservices, serverless programming, and cloud computing

WOW! eBook
www.wowebook.org

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

WOW! eBook
www.wowebook.org

Index

.

.NET Core 3
  reference link 36
.NET Core versions
  targeting, in .csproj files 97, 98

A

Active Server Pages 14
Amazon Relational Database Service (Amazon RDS) 436
Amazon Web Services (AWS)
  about 427, 495
  applications, deploying 430, 431, 432, 433
  applications, running on 445, 446, 447, 448, 449, 450, 451, 452, 454, 455, 456, 458
  logging in 505, 507
application images
  publishing, to Docker Hub 493
applications images
  publishing, to Docker Hub 491
applications
  configuring 188
  deploying, in Amazon Web Services (AWS) 430, 431, 432, 433
  deploying, in AWS Elastic Beanstalk 433, 434, 435, 436, 438, 439, 440, 441, 442, 443, 444, 446
  deploying, in Microsoft Azure 458, 459, 460, 461
  deploying, in Microsoft Azure App Services 462
  deploying, into Docker containers 482, 483, 484, 485, 486, 488, 489, 491
  deploying, on multiple environments 221, 223, 226
  hosting 428, 429
  running, on Amazon Web Services (AWS) 445, 446, 447, 448, 449, 450, 451, 452, 454, 455, 456, 458
ASP.NET Core 3 applications
  .NET Generic Host, working with 102
  about 99
  components, deciding 290
  configuring 517, 519, 520, 522, 524
  creating, in Linux 54
  creating, in Visual Studio 2019 36, 41
  creating, in Visual Studio Code 52, 54
  creating, via command line 41, 43
  cross-cutting concerns, identifying 291
  custom metrics, adding 525
  distribution for layers 290
  Docker environments, monitoring 509
  layering 288
  logging in 496, 497
  monitoring 508
  monitoring, in Amazon Web Services 528, 530, 532
  monitoring, in Microsoft Azure 512, 514, 516
  on-premises monitoring 509
  Program class, working with 100, 101
  project structure, preparing 105, 106
  required layers, determining 289
  rules, determining for interactions between layers 290
  Startup class 100
  Startup class, working with 103, 104
ASP.NET Core 3
  cache in-memory, reference link 244
  endpoint routing, using 137, 138
  features 16, 17, 19
  performance 22
  reference link 42
  scalability 22
  selecting 24, 25

WOW! eBook
www.wowebook.org

technology limitations 23
ASP.NET Core state management
about 241
client-state management options 242
server-based state management options 244
ASP.NET Core web API help pages
creating, with OpenAPI 324, 325, 326, 327, 328
creating, with Swagger 324, 325, 326, 327, 328
ASP.NET Web Forms 14
ASP.NET Web Forms applications 24
ASP.NET Web Pages applications 24
ASP.NET
history 14, 15
Atomicity, Consistency, Isolation, and Durability (ACID) 352
authentication
basic user forms authentication, adding 369, 373, 374, 376
external provider authentication, adding 377, 379
implementing 355, 356, 357, 359, 362, 365, 368
two-factor authentication 381, 383, 384, 386, 387, 389, 390
two-factor authentication, working with 380
authorization
implementing 399, 400, 402, 403, 404, 406, 407
in Postman, reference link 324
AWS Elastic Beanstalk
applications, deploying 433, 434, 436, 438, 439, 440, 441, 442, 443, 444
reference link 434
AWS Toolkit, for Visual Studio 2019
installation link 445
Azure Container Services 22
Azure DevOps build pipeline
creating 83, 84, 85, 86
Azure DevOps project
creating 62, 64
Azure DevOps release pipeline
creating 87, 88
Azure DevOps subscription
creating 62, 64
Azure DevOps
features 61
reference link 61
reference link, for organizing work 67
using, for continuous deployment 61
Azure Kubernetes Services 22
Azure Repos
continuous integration with 469, 470, 471, 472

B

behavior-driven development (BDD) 282
Blazor
about 17
reference link 203
branching strategies
reference link 76
bugs 69
build pipeline 60
bundling and minification
solutions 159
using 158, 159
working with 159, 160, 161, 162, 163

C

C# Interactive 56, 57
C# Razor components
about 19
used, for client-side development 203, 204, 206, 207
clickjacking
about 422
mitigating 422, 423, 424
vulnerability, example 422
client-side development
C# Razor components, using 203, 204, 206, 207
client-state management options
about 242
cookies 242
hidden fields 242
query string 243
query string usage 243
code-behind files 14
Configuration API
reference link 191
confused deputy attack 417
containers

WOW! eBook
www.wowebook.org

working with 22
continuous deployment (CD)
about 59, 60
Azure DevOps, using 61
continuous integration (CI)
about 59, 60
Azure DevOps, using 61
with Azure Repos 469, 470, 471, 472
controllers 231
cookie stealing
about 412
preventing 412, 413
cross-platform support 19, 20
Cross-Site Redirects (XSR) 414
Cross-Site Request Forgery (XSRF/CSRF)
about 417
domain referrers 418
example 417, 418
limitations 420
preventing 418
user-generated tokens 419
Cross-Site Scripting (XSS)
about 410
preventing 411

D

Data Annotations
localizing 184, 185, 186, 187, 188
data relationships
about 346
foreign key 347
many-to-many relationships 349, 350
one-to-many relationships 348
one-to-one relationships 348
primary key, designing 346
database querying, with LINQ
for all items 351
for filtered items 351
for one item 350
dedicated layouts
creating, for multiple devices 232, 233, 235
dependency injection (DI)
about 105, 194
concepts, implementing 194
Development (Dev) 495
DI container
using, to encourage loose coupling 122
dispatcher 137
distributed session providers
about 173
examples 173
Docker containers
applications, deploying into 482, 483, 484, 485, 486, 488, 489, 491
Docker Enterprise Edition
installation link 484
Docker environments 512
Docker Hub
about 491
application images, publishing 491, 493
reference link 483
URL 492
Docker
reference link 483
Don't Repeat Yourself (DRY) principle 232

E

eavesdropping
about 413
preventing 413, 414
Elastic Beanstalk
applications, deploying 435
email confirmation functionality
building 147, 148
email confirmation
by user 149, 150, 151, 152, 153
email service
adding 188, 189, 190
configuring 190, 191, 192, 193, 194
endpoint routing
used, for ASP.NET Core 3 137, 138
Entity Framework (EF) 330, 444
Entity Framework Core 3
about 330
connection, establishing 333, 335, 336
data, creating 345
data, deleting 345, 346
data, reading 345
data, updating 345
foreign keys, defining via Data Annotations 336,

WOW! eBook
www.wowebook.org

337
migrations feature, using 340, 341, 342, 343, 344
primary keys, defining via Data Annotations 336, 337
reference link 331
Entity Framework Data Annotations
reference link 339
error handling
adding, to Tic-Tac-Toe application 138, 140, 141, 143
Event Tracing for Windows (ETW)
about 496
reference link 509
exception handling middleware 128
external provider authentication
adding 377, 379

F

fail fast 60
feature branches
using 77, 78
Fiddler
URL 312
file logging provider 215
forgotten password mechanism
adding 390, 393, 395, 396, 398, 399

G

Git
changes, merging 79, 82
conflicts, resolving 79, 82
feature branches, using 77, 78
reference link 71
using, as version control system (VCS) 71, 73, 74, 76
GitHub Actions
URL 469
globalization
about 174
applying, for multi-lingual user interfaces 174
concepts 174, 176, 177, 178, 179, 180, 181
reference link 174
group 413
gRPC Service template 19
gRPC template 16

H

HATEOAS-style web APIs
building 321, 322, 323
heavy coupling 229
Hibernate Query Language (HQL) 417
hubs 208
Hypermedia as the Engine of Application State (HATEOS) 321
Hypertext Transfer Protocol (HTTP) 243

I

in-memory session provider 169, 170, 171, 172
integrated development environment (IDE) 27
integration tests
about 232
adding 286, 287
applying 270
creating 277
Internet Information Server (IIS) 423
Internet Information Services (IIS) 429, 498
IT Operations 495

J

JavaScript, using for client-side development
about 146, 147
email confirmation functionality, building 147, 148
email confirmation, by user 149, 150, 151, 152, 153
XMLHttpRequest (XHR), using 153, 154, 155, 156, 157
JavaScript
client-side development 153
JSON hijacking
about 420
preventing 420

L

Language-Integrated Queries (LINQs) 57
layout page
updating 115, 116, 117, 118
used, for enhancing web pages 112, 113, 114
LINQ Query Operations

WOW! eBook
www.wowebook.org

reference link 351
LINQPad
about 56
reference link 351
Linux Container (LXC) 22
Linux Ubuntu installation
download link 50
Linux
ASP.NET Core 3 application, creating 54
Visual Studio Code, installing 49, 52
localization
applying, for multi-lingual user interfaces 174
concepts 174, 176, 177, 178, 179, 180, 181
reference link 174
Log4net 496
logging functionality
used, for monitoring and supervision purposes 209, 211, 213, 215, 217, 221
LTTng
reference link 509

M

mean time to repair (MTTR) 277, 495
message replay
about 413
preventing 413, 414
message tampering 413
method injection
about 194
using 194, 195, 197, 199, 200
microservice architecture
about 21
containers, working with 22
Microsoft Azure App Services
applications, deploying 462
database, connecting 474
instance, running 462, 463, 464, 465, 466, 467
logging in 498, 499, 502, 503, 504
reference link 462
Microsoft Azure
application, deploying 460
applications, deploying 458, 459, 461
code, publishing 468
database, connecting 472, 473, 474, 475, 476, 477, 478
logging in 497
subscription link 458
URL 459
Microsoft Intermediate Language (MSIL) 416
Microsoft SDL
about 411
reference link 411
Microsoft.AspNetCore.App metapackage
using 98, 99
model binding 420
Model View Controller (MVC) pattern
about 229
controllers 231
integration tests 232
models 230
unit tests 231
views 231
models 230
multi-lingual user interfaces
globalization, applying 174
localization, applying 174

N

Node Package Manager (NPM)
about 158
URL 112
used, for enhancing web pages 112, 113, 114
noughts and crosses game 91
NSwag 18
NuGET packages 18

O

object-relational mapping (ORM)
about 230, 330
using 416, 417
open redirects
about 414
example 414
preventing 415
OpenAPI
used, for creating ASP.NET Core web API help pages 324, 325, 326, 327, 328
operations (Ops) 495
over-posting
about 420

WOW! eBook
www.wowebook.org

mitigating 421
vulnerability, example 421
OWASP Top 10
reference link 354

P

partial views
using 245, 253
password reset mechanisms
adding 390, 391, 395, 396, 398
Persist Security Info default value
using, in SQL connection strings 416
plain old CLR objects (POCO) 190, 211
Platform as a Service (PaaS) 429
Postman
URL 312
product backlog items (PBI) 65
proof of concept (PoC) 429
public key infrastructure (PKI) authentication 355

Q

queries
working with 350

R

read-eval-print-loop (REPL) tool 56
regular expressions (regex) 236
relational database management systems (RDBMS) 331
release pipeline 60
Remote Procedure Call (RPC) 296
response compression middleware 128
REST-style web APIs
building 312, 313, 314, 315, 316, 317, 318, 319, 320, 321
RPC-style web APIs
building 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312
runtime compilation 17

S

scoped injection 123
screen of death (SOD)
about 424
mitigating 425
vulnerability, example 425
scrum process 66, 67, 69, 70
search engine optimization (SEO) 234
Secure Sockets Layer (SSL) 413
Security Development Lifecycle (SDL) 411
separation of concerns (SoC) 229
sequence diagram 94
Serilog 496
server-based state management options
about 244
application state 244
session state 244, 245
service-level agreements (SLAs) 429
services
configuring 188
session
using 168
SignalR
about 207
using, with Razor components 208, 209
using, with server-side Blazor 208
working with 207
Simple Object Access Protocol (SOAP) 296
Single Responsibility Principle (SRP) 229
Single Sign-On (SSO) authentication 355
singleton injection 123
SoapUI
reference link 312
solutions, for handling logging
Azure Application Insights 497
Azure Application Service diagnostic 497
standard file logging 497
SQL connection strings
Persist Security Info default value, using 416
preventing 416
SQL injection
about 415
preventing 415
SQL Server Management Studio (SSMS) 442, 444
SQLite 331
starter workflows
reference link 469

WOW! eBook
www.wowebook.org

static files middleware 128
Swagger
used, for creating ASP.NET Core web API help pages 324, 325, 326, 327, 328

T

Tag Helpers
about 231
using 245, 260, 262, 263, 265, 266
telemetry
used, for monitoring and supervision purposes 209, 211, 213, 215, 217, 219, 221
templates
reference link 24
test-driven development (TDD) 44, 282
Tic-Tac-Toe application
about 145
communication middleware, creating 127, 130, 132
communication middleware, working with 127, 128, 130
error handling, adding 138, 140, 141, 143
routing, using 134, 136
static files, working with 132, 133
URL redirection 134
URL redirection, using 134, 136
URL rewriting, using 134, 136
Tic-Tac-Toe demo application
optimizing, for mobile devices 236, 238, 239, 241
preview 91
Tic-Tac-Toe game
.NET Core versions, targeting in .csproj files 97, 98
building 92, 93
feature, conceiving 93, 95, 96
feature, implementing 93, 95, 96
home page, creating 107, 110, 111
Microsoft.AspNetCore.App metapackage, using 98, 99
Tic-Tac-Toe user registration page
creating 119, 121, 122
Tic-Tac-Toe user service
creating 122, 123, 125, 126
DI container, using to encourage loose coupling 122
transactions
using 352, 353
transient injection 123
two-factor authentication
working with 380, 381, 383, 387, 389, 390

U

UI redress attack 422
UI redressing 422
unit tests
about 231
adding 282, 283, 285
applying 270
creating 277, 278, 281
URL rewriting middleware 135
user cache management
using 168
user experience (UX) 115, 245
user forms authentication
adding 369
implementing 374, 376
user interface (UI) 245
user stories 93

V

view components
about 231
using 245, 254, 256, 257, 259, 260
view engines
using 270, 272, 273, 274, 276, 277
view localizer
using 181, 182, 183
view pages
for scaffolding features 246
using 245, 246, 247, 249, 251, 252
views 231
Virtual Machine (VM) 483
Visual Studio 2019 Community Edition, installation
about 28, 30, 31
ASP.NET Core 3 application, creating 36, 41
ASP.NET Core 3 application, creating via command line 43
express installation 28
offline installation 29

WOW! eBook
www.wowebook.org

reference link 29
Visual Studio 2019
autos 47
breakpoints 45, 46
call stack 46
code refactoring feature 33
exploring 35, 36
live code analysis feature 33
locals 47
options 33
reference link 28
used, for debugging 43, 44
using, as development environment 27
Watch Panes 47
Visual Studio Code, user interface
activity bar 48
editor groups 48
panels 48
side bar 48
status bar 48
Visual Studio Code
ASP.NET Core 3 application, creating 52, 54
ASP.NET Core 3 application, creating in Linux 54
installing, on Linux 49, 52
reference link 49
URL, for installing on Linux 51
using, as development environment 47
Visual Studio Team Services (VSTS) 60

W

WCF Services 24
web API
authorization options 324
best practices 294, 295
concepts, applying 294, 295
creating, styles 295
securing 323, 324
web application
optimizing 158, 159
separating, into multiple areas 266, 267, 269, 270
Web Deploy tool
using, for deployment 479, 480, 481, 482
web pages
enhancing, with layout pages 112, 113, 114
enhancing, with NPM 112, 113, 114
WebSockets
for real-time communication scenarios 163
reference link 163
working with 164, 165, 166, 168
Windows Forms (WinForms) 14
work item types (WITs) 65
work items
work, organizing 64
Worker Service template 19
workflows 93
World Wide Web (WWW) 207

X

XMLHttpRequest (XHR)
about 153
using 153, 154, 155, 156, 157
xrdp
installing 49

WOW! eBook
www.wowebook.org